Embora a programação estandardizada esteja ao alcance de todos, mais cedo ou mais tarde, os utilizadores de microcomputadores terão de elaborar os seus próprios programas. Para essa eventualidade e para melhor compreensão do computador, o autor apresenta, neste acessível guia, a noção de programa e os passos essenciais da programação. Incluindo testes, programas, exercícios e um glossário de termos técnicos, a presente obra constitui relevante auxílio para o leitor que pretende iniciar-se no universo da microinformática.

EDITORIAL PRESENÇA / MARTINS FONTES

introdução à Programação de Microcomputadores

130

CULTURA E TEMPOS LIVRES

1. ABC do Xadrez, *Petar Trifunovitch* e *Sava Vukovitch*
4. ABC do Bridge, *Pierre Jais* e *H. Lahana*
5. Guia Prático de Fotografia, *W. D. Emanuel*
6. ABC do Judo, *E. J. Harrison*
7. Como Fazer Cinema, *Paul Petzold*
8. Bridge Moderno, *Pierre Jais* e *H. Lahana*
9. Fotografia — Técnicas e Truques I, *Edwin Smith*
10. Estilos de Mobiliário, *A. Aussel*
11. Fotografia — Técnicas e Truques II, *Edwin Smith*
12. A Pesca Submarina, *António Ribera*
13. Teoria dos Finais de Partida, *Yuri Averbach*
14. Aprenda Rádio, *B. Fighiera*
15. Guia do Cão, *Louise Laliberté-Robert* e *Jean-Pierre Robert*
16. ABC do Aquário, *Anthony Evans*
17. Iniciação à Electricidade e Electrónica, *Fernand Huré*
18. Os Transístores, *Fernand Huré*
19. Karaté I, *Albrecht Pflüger*
20. Iniciação ao Radiocomando dos Modelos Reduzidos, *C. Péricone*
21. Construa o seu Receptor, *B. Fighiera*
22. Montagens Electrónicas, *B. Fighiera*
23. O Berbequim Eléctrico, *Villy Dreier*
24. Cactos, *J. Nilaus Jensen*
25. Iniciação à Alta Fidelidade, *Peter Turner*
26. O Aquário de Água Doce, *Paulo de Oliveira*
27. ABC do Ténis, *Fonseca Vaz*
28. Karaté II, *Albrecht Pflüger*
29. ABC da Criação de Canários, *Curt Af Enehjelm*
30. Ginástica Feminina, *Sonja Helmer Jensen*
31. Cartomancia, *Rhea Koch*
32. Calculadoras Electrónicas de Bolso, *E. Dam Ravn*
33. O Pastor Alemão, *Gilles Legrand*
34. Xadrez — Teoria do Meio Jogo, *I. Bondarevsky*
35. Manual do Super 8-I, *Myron A. Matzkin*
36. ABC da Criação de Periquitos, *Cyril H. Rogers*
37. O Livro dos Gatos, *Bärbel Gerber* e *Horst Bielfeld*
38. Manual do Super 8-II, *Myron A. Matzkin*
39. ABC do Mergulho Desportivo, *Walter Mattes*
40. Circuitos Integrados/Aplicações Práticas, *F. Bergtold*
41. A Apicultura, *H. R. C. Riches*
42. ABC do Cultivo das Plantas, *H. G. Witham Fogg*
43. ABC da Criação de Pombos, *Kai R. Dahl*
44. Construção de Caixas Acústicas de Alta Fidelidade, *R. Brault*
45. Raças de Canários, *Klaus Speicher*
46. Jogos de Cartas, *Graciano Dolma*
47. Cocker Spaniels, *H. S. Lloyd*
48. ABC da Caça, *Fabián Abril*
49. Aprenda Televisão, *Gordon J. King*
50. Iniciação à Pesca, *Juan Nadal*
51. Basquetebol, *Marius Norregard*
52. Cães de Caça, *Santiago Pons*
53. Aprenda Electrónica, *T. L. Squires* e *C. M. Deason*
54. A Avicultura, *Jim Worthington*
55. A Produção de Coelhos, *P. Surdeau* e *R. Henaff*
56. ABC dos Computadores, *T. F. Fry*
57. Natação para Crianças, *John Idorn*
58. O Boxer, *Anni Mortensen*
59. Voleibol, *Ole Hansen* e *Per-Göran Persson*
60. Iniciação à Vela, *Donald Law*
61. ABC da Filatelia, *Jacqueline Caurat*
62. A Pesca à Beira-Mar, *J.-M. Boëlle* e *B. Doyen*
63. Enxerto de Árvores de Fruto, *Alejo Rigau*
64. A Cultura do Morangueiro, *Luís Alsina Grau*
65. Emissores-Receptores (Walkies-Talkies), *P. Duranton*
66. Iniciação à Fotoelectrónica, *Heinz Richter*
67. Doces e Conservas de Fruta, *Robin Howe*
68. A Criação de Hamsters, *C. F. Snow*
69. A Criação de Porcos, *Roy Genders*
70. Calendário do Horticultor, *Luís Alsina Grau*
71. Jogos Electrónicos, *F. G. Rayer*
72. Cultivo de Cogumelos e Trufas, *Alejo Rigau*
73. Aprenda Televisão a Cores, *Gordon J. King*
74. Gravação em Fita Magnética, *Ian R. Sinclair*
75. Poda de Árvores e Arbustos, *Roy Genders*
76. Como Treinar o Seu Cão, *E. Fitch Daglish*
77. Instrumentos de Medida e Verificação, *Heinrich Stöckle*
78. A Criação de Caracóis, *Matías Josa*
79. Rádio — Fundamentos e Técnica, *Gordon J. King*
80. Como Fazer Gelados, *Sylvie Thiébault*
81. Iniciação à Jardinagem, *Noel Clarasó*
82. A Congelação dos Alimentos, *Suzanne Lapointe*
83. Windsurf — Prancha à Vela, *Ernstfried Prade*
84. Raças de Cães, *O. Hasselfeldt*
85. Rummy e Canasta, *Claus D. Grupp*
86. A Encadernação, *Annie Persuy*
87. Aprenda Electricidade, *Heinz Richter*
88. Taxidermia, Embalsamamento de Aves e Mamíferos, *Harry Hjortaa*
89. Jogging — Correr para Manter a Forma, *Werner Sonntag*
90. ABC da Cozinha Chinesa, *Sonya Richmond*
91. Jogos T. V., *C. Tavernier*
92. Amplificadores de Som, *Richard Zierl*
93. O Livro do Poker, *Claus D. Grupp*
94. Aprenda a Desenhar, *Rose-Marie de Prémont* e *Nicole Philippi*

95. O Minitrampolim na Escola, *Sonja Helmer Jensen* e *Klaus Dano*
96. Jogos de Luzes e Efeitos Sonoros para Guitarras, *B. Fighiera*
97. O Cultivo do Tomate, *Louis N. Flawn*
98. Pilhas Solares, *F. Juster*
99. A Criação Doméstica de Coelhos, *C. F. Snow*
100. Iniciação ao Futebol, *Wieland Männle* e *Heinz Arnold*
101. Horóscopos Chineses, *Georg Haddenbach*
102. Guia Prático de Marcenaria, *Charles H. Hayward*
103. Andebol, *Fritz* e *Peter Hattig*
104. Dispositivos Anti-Roubo, *H. Schreiber*
105. Perus, Pintadas e Codornizes, *Jerome Sauze*
106. Crepes — Doces e Salgados, *Florence Arzel*
107. Aperitivos e Entradas, *Myrette Tiano*
108. Ténis de Mesa, *Leslie Woollard*
109. Aprenda Surf, *Rick Abbott* e *Mike Baker*
110. Futebol — Técnica e Táctica, *Kurt Lavall*
111. A Vaca Leiteira, *C. T. Whittemore*
112. O Cubo Mágico, *Josef Trajber*
113. O Perdigueiro Português, *José M. Correia*
114. Pizzas e Massas à Italiana, *Marieanne Ränk*
115. O Cubo para quem o já faz, *Josef Trajber*
116. A Pirâmide Mágica, *M. Mrowka* e *W. J. Weber*
117. Gansos e Patos, *Marie Mourthe*
118. Iniciação ao Kung Fu, *A. P. Harrington*
119. Electrónica e Fotografia, *Hanns-Peter Siebert*
120. O Livro da Fortuna, *Douglas Hill*
121. Construção de um Alimentador de Corrente. *Waldemar Baitinger*
122. Hóquei em Patins, *Francisco Velasco*
123. Técnicas de Tiro, *Anton Kovacic*
124. Aprenda a Tricotar. *Uta Mix*
125. ABC da Patinagem, *Christa-Maria* e *Richard Kerler*
126. A Pesca e os seus Segredos, *Armand Deschamps*
127. O Osciloscópio, *R. Rateau*
128. Guia Prática da Banda do Cidadão, *Jean-Michel Normand*
129. Sumos e Batidos, *Manfred Donderski*
130. Introdução à Programação de Microcomputadores, *Peter C. Sanderson*

PETER C. SANDERSON

INTRODUÇÃO À PROGRAMAÇÃO DE MICROCOMPUTADORES

Título original:
INTRODUCTION TO MICROCOMPUTER PROGRAMMING

Tradução de Conceição Jardim e Eduardo Nogueira
Foto da Capa (Microcomputador *New Brain*) gentilmente cedida
por *Landry Engenheiros Consultores, Lda — Lisboa*

PREFÁCIO

O desenvolvimento de sistemas computadores baseados em pastilhas microprocessadoras destruiu a barreira de custos que anteriormente impedia o uso mais generalizado do computador. É hoje possível utilizar computadores numa vasta gama de campos, em particular em pequenos negócios, por professores e até em «hobbys». No entanto, se bem que os sistemas em causa sejam baratos, são inúteis quando não se dispõe de programas apropriados. A encomenda de um programa especial para uso individual é cara; um conjunto de programas comerciais é provavelmente mais caro do que o próprio sistema microcomputador. Por outro lado, o uso dos programas «standard» confina o utilizador à camisa de forças criada pelos conceitos do fornecedor. Muitas aplicações para comércio e para hobbys são tão individuais que um conjunto de programas standard se torna completamente inadequado. Mais tarde ou mais cedo, a maior parte dos utilizadores de sistemas microcomputadores terão de pensar em criar os seus próprios programas a fim de poderem aproveitar todas as qualidades do sistema que tiverem adquirido.

Este livro é uma introdução simples à programação para utilizadores de microcomputadores, tanto no caso de utilizadores comerciais como de «hobbyists» ou professores. Parte-se do princípio de que o leitor tem algum conhecimento das funções de um computador, mas não é de modo algum essencial um conhecimento matemático ou uma familiaridade especiais com a construção electrónica ou funcionamento dos computadores para conseguir

compreender o texto. Não é igualmente necessário qualquer conhecimento prévio de programação. Este livro constitui essencialmente um guia prático de programação, e um manual de auto-instrução. Os exemplos são escolhidos em campos familiares, considerando-se que estes exemplos e exercícios serão aplicados ao sistema microcomputador do leitor. Fornecem-se soluções para cada um dos exercícios indicados.

Os capítulos sobre a linguagem Basic indicam as características que constituem um núcleo comum e podem ser encontradas na grande maioria implementações para microcomputadores. O capítulo 7 é dedicado a algumas das variantes da linguagem Basic vulgarmente presentes em configurações microcomputadoras. Os capítulos sobre linguagem Assembler e sobre o código-máquina tratam das características das quatro pastilhas microprocessadoras mais vulgarmente usadas. Dado que programar é muito mais do que simplesmente codificar nas expressões utilizadas pela linguagem considerada, dedica-se um capítulo inicial às actividades anteriores à codificação, e o capítulo final ao desenvolvimento de programas e ao seu ensaio.

Gostaria de exprimir os meus agradecimentos a todos os fabricantes e fornecedores que me cederam materiais sobre as suas versões da linguagem Basic, essenciais para escrever o capítulo 7 deste livro. Gostaria de mencionar em particular Anne Patricia, cuja vida terminou tragicamente em Outubro de 1978, e que me sugeriu a realização deste livro, que saiu das muitas conversas entre nós sobre a programação de computadores. De certo modo, posso dizer que esta obra é a minha homenagem à sua memória.

Gostaria finalmente de registar aqui a minha profunda gratidão a Kathlyn Bell pela sua ajuda na dactilografia, à minha filha Júlia, que me estimulou quando faltava a inspiração, e ainda à equipa da casa editora que muito me auxiliou.

Peter C. Sanderson

1

INTRODUÇÃO À PROGRAMAÇÃO DE COMPUTADORES

Um sistema computador que não saibamos programar pode ser comparado a um instrumento que não possua um manípulo. Se você perdeu muitas horas montando e ensaiando o seu próprio «kit», mais tarde ou mais cedo desejará escrever programas para ele.

Um *programa* é uma série de instruções que permite ao computador realizar a tarefa que você pretende, quer se trate de apresentar um jogo no «écran» da televisão ou de calcular juros a pagar. É escrito numa linguagem que o sistema computador é capaz de compreender. Quando se escrevem programas de computador, é necessário estar-se familiarizado com o código de programação (ou *linguagem*) particular com que se está a trabalhar. Isto assemelha-se um pouco a actividades, como a leitura de mapas, em que é necessário familiarizar-nos com o código dos símbolos neles usados e as cores dos contornos.

A palavra «programa» é a primeira de muitos termos técnicos que serão utilizados neste livro. É infelizmente inevitável o emprego de algum jargão numa obra sobre computadores. Neste capítulo as palavras não familiares serão apresentadas em itálico da primeira vez que surgem, sendo imediatamente definidas. No final do livro encontrará um glossário alfabético dos termos especiais de que necessitará para ler este livro ou qualquer outro sobre computadores.

Se está a usar o seu sistema microcomputador como estudante, necessitará quase de certeza de escrever os seus próprios programas. Se o usa para o seu hobby verificará mais tarde ou mais cedo que os programas «standard» fornecidos pelo fabricante ou obtidos em revistas da especialidade não se adaptam exactamente àquilo que você pretende. Os sistemas domésticos de contabilidade e de orçamento variam de uma casa para outra, pelo que os programas estandardizados dificilmente concordarão com as suas necessidades específicas. Se está a usar um sistema microcomputador para controlar algum aparelho externo não encontrará provavelmente um sistema capaz de o fazer do melhor modo, quer o aparelho externo seja um sintetizador musical, um aquecimento central ou um sistema de comboios em miniatura. Os jogos de computador que se podem adquirir no comércio acabam sempre por se tornar aborrecidos ou demasiado simples, e você desejará então introduzir neles as suas próprias variantes ou conceber um novo jogo, para o que necessitará evidentemente de conceber os programas necessários.

A programação de um microcomputador é uma tarefa essencialmente semelhante à programação de um computador convencional de grandes dimensões ou dos computadores de dimensões médias também chamados *minicomputadores*. Com efeito, se utilizar certas linguagens de programação para o seu microcomputador, verificará que os seus programas poderão também ser utilizados por computadores convencionais como o IBM 370/158. Como é óbvio, o trabalho preparatório feito antes de escrever as instruções na linguagem de programação conveniente é igual para todos os tipos e dimensões de computadores.

Todos os computadores seguem obedientemente as instruções contidas no programa. Não são telepatas e não sabem se você se esqueceu de lhes dar uma instrução porque, para a mente humana, ela parece demasiado óbvia. Se se esquecer de indicar a alguns computadores quando devem parar no fim de um programa, ou de lhes

«dizer» que não devem realizar uma divisão quando um dos números nela envolvidos é zero, o sistema começará a apresentar na saída respostas sem qualquer sentido. Quando se escrevem programas de computador, devem-se abandonar todas as pretensões intelectuais e tentar ver as instruções ao mesmo nível a que a máquina as vê. Todas as instruções do seu programa serão meticulosamente obedecidas, e o computador nunca tentará descobrir se são lógicas ou englobam todas as possibilidades presentes.

Nestas condições, a programação pode ser um trabalho frustrante que exige uma enorme atenção aos pormenores. No entanto não é desmedidamente difícil; não o é mais do que aprender a tocar algumas melodias simples num piano a partir da notação musical convencional. Muitos óptimos profissionais de programação partiram de conhecimentos matemáticos nulos. O número rapidamente crescente de «hobbyists» que utilizam computador demonstra que não há nada de verdadeiramente «formidável» na escrita de programas, e que esta pode até acrescentar algum desafio extra ao seu hobby.

A regra mais importante na concepção de programas com êxito consiste em evitar escrever à pressa as várias instruções sem ter realizado todo o trabalho preparatório necessário, que será descrito no resto deste capítulo. Há uma distinção fundamental entre escrever as instruções (ao que no mundo dos computadores se chama *codificar*) e o verdadeiro trabalho de concepção do programa. Excepto no caso dos programas mais simples, a codificação corresponde a menos de um quarto do trabalho necessário. Tal como na decoração de uma casa ou de uma sala, quanto mais tempo for gasto na preparação, melhor, mais elegante e durável será o resultado. Tal como a pintura sobre uma superfície não preparada depressa necessita de ser substituída, também os programas escritos à pressa depressa obrigam a alterações drásticas. Aliás, há até grandes probabilidades de que nem sequer consigam ser executados completamente pela máquina.

Existem cinco fases principais na preparação de um programa, antes de o introduzir pelo teclado ou pelos interruptores do microcomputador no código ou linguagem apropriados.

1. Verificar se o problema pode ser tratado pelo seu microcomputador.
2. Definir completamente o problema.
3. Considerar os vários métodos possíveis de o solucionar.
4. Dividir o problema em fases pequenas, que possam ser representadas por instruções na linguagem de programação, e exprimi-las numa forma visual (chama-se a isto fazer um *fluxograma*, e será definido com maior rigor mais adiante).
5. Documentar as indicações necessárias, de modo a facilitar a utilização do programa a partir dessa documentação, que também o auxiliará quando quiser ampliar ou alterar o programa original.

Estas fases serão descritas pormenorizadamente no resto deste capítulo.

Verifique se o problema pode ser resolvido pelo computador

A maior parte dos problemas podem ser resolvidos num dado computador se puderem ser expressos numa série lógica de passos bem definidos. Em teoria, qualquer problema que não contenha elementos irracionais ou intuitivos pode ser programado para realização por computador, se bem que em alguns casos o programa será demasiado complexo para poder ser escrito num tempo razoável. Pode-se conceber um programa para analisar dados do passado e prever o possível vencedor de um jogo de futebol ou de uma corrida de cavalos, mas não pode culpar o programa se o microcomputador não conseguir acertar pois no caso das competições desportivas estão sempre envolvidos elementos que desafiam a lógica.

Alguns problemas, que em teoria podem ser resolvidos por um computador, não poderão ser resolvidos

pelo equipamento concreto que você possui, ou só o podem ser com bastante dificuldade. Se o seu sistema não possui um teclado, é melhor evitar aplicações em que seja necessário introduzir letras do alfabeto, e como é óbvio não é possível elaborar «displays» gráficos se só dispuser de um display LED. Se deseja usar um grande ficheiro de dados no seu programa, pode ser muito demorado e trabalhoso ter de mudar muitas cassettes de fita durante o funcionamento do programa.

Você pode achar que o seu programa é demasiado grande para *a memória* do equipamento. Facilmente descobrirá a dimensão da memória consultando o manual que acompanha o sistema, e se você próprio o tiver montado saberá certamente quantas memórias lhe ligou. Depressa se habituará a avaliar o tamanho de um programa logo no início da sua concepção. Deve decidir então se o programa deve ser abandonado (ou adiado até comprar mais memórias) ou se pode ser subdividido numa série de problemas mais curtos, de tal modo que lhe permita utilizar os dados produzidos por um programa como entrada para um novo programa que se lhe siga.

Como se se tratasse de um pianista inexperiente tentando tocar uma sonata de Brahms, você pode decidir que, apesar de o programa poder ser feito, ser-lhe-á muito difícil concebê-lo até ter ganho mais experiência. Esta dificuldade não o deve levar a pensar que é um incapaz neste campo. Verificou-se nos Estados Unidos que a produção normal *correcta* em código-máquina (a forma mais difícil de linguagem de programação, explicada no capítulo seguinte) de um programador é de apenas dez instruções por dia. Nestas condições, você pode não dispor do tempo necessário para programar um problema complexo.

Se você tem um pequeno negócio e utiliza microcomputadores, deve estudar a potencial poupança decorrente da introdução de um dado sistema microcomputador antes de se preocupar com a programação. Estas considerações determinarão aliás os projectos que você irá programar primeiro.

Definição do problema a programar

Os problemas mais fáceis de definir para programação num microcomputador são os numéricos. Existem muitos *algoritmos* (regras para resolução de problemas específicos, muitas vezes aplicadas a uma linguagem de programação) publicados para problemas numéricos, que podem necessitar apenas de pequenas alterações para poderem ser aplicados no seu sistema microcomputador. Quer os use quer conceba de uma ponta à outra o seu programa, deve verificar se:

— tem os necessários cuidados no caso de qualquer número ou resultado intermédio ser provavelmente zero ou negativo;
— nenhum número ou resultado intermédio tende a ser demasiado grande ou pequeno para o seu sistema (o manual indicará a gama de números que podem ser representados);
— pode obter o rigor que pretende dos resultados;
— está a confirmar devidamente os números que fornece ao computador através de teclas ou interruptores; é muito fácil cometer erros deste modo, e nem sempre é fácil detectá-los no momento em em que são feitos.

Se está a programar um problema em que trabalha com sinais de interrupção exteriores ao microcomputador, como os vindos de uma instalação de comboios em miniatura, deve verificar se se encontram definidos os requisitos de tempo convenientes no programa. Deve ainda pensar cuidadosamente naquilo que irá incluir no programa prevendo o caso de condições não habituais ou de erro.

Os programas domésticos necessitam de ser definidos tão rigorosamente como os problemas matemáticos. Deve tentar visualizar as diversas maneiras de a sua família poder fornecer dados pelo teclado. Se está a trabalhar num calendário para a aplicação diária, por exemplo, deverá decidir se aceita 11 Jun, 11 Junho ou 11.6 como entradas igualmente possíveis, e se aceita o algarismo «0» e a letra «O» como intermutáveis. Pode estar a introduzir controlos a mais sobre os dados do teclado, mas é melhor prever muitas variantes nas primeiras fases da concepção de programas.

Tanto em programas domésticos como de negócios você irá provavelmente confrontar um ficheiro em cassettes de fita com informações fornecidas por teclado. Um exemplo típico desta aplicação seria a conciliação de créditos e débitos que você apresenta ao computador em cassette com entradas vindas de um extracto de conta enviado pelo banco, que você apresenta ao computador através do teclado. Deve programar uma acção correcta quando por exemplo se atinge o final do ficheiro em cassette sem ter ainda introduzido no computador todos os dados vindos do banco, ou quando estes acabam antes de o computador processar todo o conteúdo da cassette. Se os depósitos e levantamentos que você faz no banco se encontram em mais do que uma cassette, você deve programar uma mensagem para que o computador peça a introdução de uma nova cassette. Neste tipo de programa você deve ser liberal ao instruir o computador para apresentar ou imprimir mensagens a fim de controlar a entrada de dados por teclas e para explicar condições pouco habituais ou erros à pessoa que está a trabalhar com a máquina. Em todos os programas, você deve evitar que o resultado seja apresentado sob a forma de uma cadeia de dígitos no caso de o seu microcomputador ser capaz de apresentar títulos e texto explicativos juntamente com os números.

Como fazer fluxogramas

Depois de ter definido o problema, devem-se considerar os possíveis métodos de solução, escolhendo finalmente um deles. Em seguida deve dividir o método em diversas fases; cada uma delas acabará eventualmente por equivaler a uma única instrução de programa, na linguagem

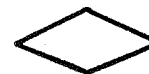
que estiver a usar. Até certo ponto, os processos de selecção de um método e da sua decomposição em instruções são complementares, dado que muitas vezes um exame detalhado de um método originalmente escolhido demonstrará que não é o mais apropriado. É então necessário começar tudo de novo, estudando um método alternativo para resolução do problema anteriormente definido.

Definiu se anteriormente o fluxograma com uma expressão dos passos de programação numa forma visual ou gráfica. Você não irá evidentemente escrever detalhadamente todas as instruções, mas sim começar por definir em geral as fases necessárias para a resolução do problema. Podemos portanto ampliar a nossa definição de fluxograma de modo a incluir uma representação visual das fases deste processo necessárias para resolver o problema no computador.

A expressão das fases da resolução de um problema numa forma visual é um valioso auxiliar da programação. Quando se escreve uma fórmula, ou se estuda uma série de instruções (como uma receita), é difícil compreender que elas são apenas uma expressão de um certo número de passos independentes que se seguem uns aos outros no tempo, pois tendemos a aprender a fórmula ou as instruções como um conjunto. No entanto, o computador só obedece a uma instrução de cada vez, pelo que o fluxograma é ideal para a representação de fases num programa de computador dado que ilustra a sequência temporal das diversas operações a realizar. Um fluxograma torna muitas vezes mais óbvios os passos que se seguem à realização de uma operação, ajudando a evitar a repetição desses passos. É mais fácil alterar um fluxograma do que as instruções de programa dadas na linguagem que o computador aceita, principalmente para um principiante.

Os fluxogramas não se usam apenas em programação de computadores. São usados numa variedade cada vez maior de aplicações, incluindo montagens mecânicas, produção química e rastreio de deficiências em maquinaria.

A forma dos «blocos» onde são escritos os passos de um fluxograma encontra-se normalizada. Neste livro só

serão utilizados os dois símbolos principais, dado que é perfeitamente possível a um amador de programação realizar todo o seu trabalho usando apenas estes. São:

O uso destes símbolos é ilustrado num simples fluxograma feito para o jogo Snakes & Ladders (figura 1.1). Este exemplo ilustra diversas características importantes da construção de fluxogramas:

1. Usam-se blocos para começo («start») e paragem («stop») do programa.

2. A quantidade de pormenores depende inteiramente de si. Neste diagrama, incluímos numa única caixa a possibilidade de cair num «snake» ou num «ladder» (I).

3. Raramente existe uma única solução correcta para um dado fluxograma ou programa. É sem dúvida possível encontrar uma solução diferente, e igualmente válida, para o jogo ilustrado. Por exemplo, no bloco G pode-se realizar a verificação « $<$ 100?» («menor do que 100?») e transformar a sua saída «sim» no trajecto principal do programa.

4. O diagrama mostra a complexidade de um problema que à primeira vista parece trivial. Mostra igualmente a importância da colocação dos blocos de decisões pela ordem correcta, de modo a evitar ter de escrever passos idênticos em muitos ramos diferentes do fluxograma ou do programa, e evitar (com sorte) combinações de acontecimentos que dêem origem a acções erradas.

Talvez já esteja convencido de que o tempo gasto na planificação de um programa sob a forma de fluxograma

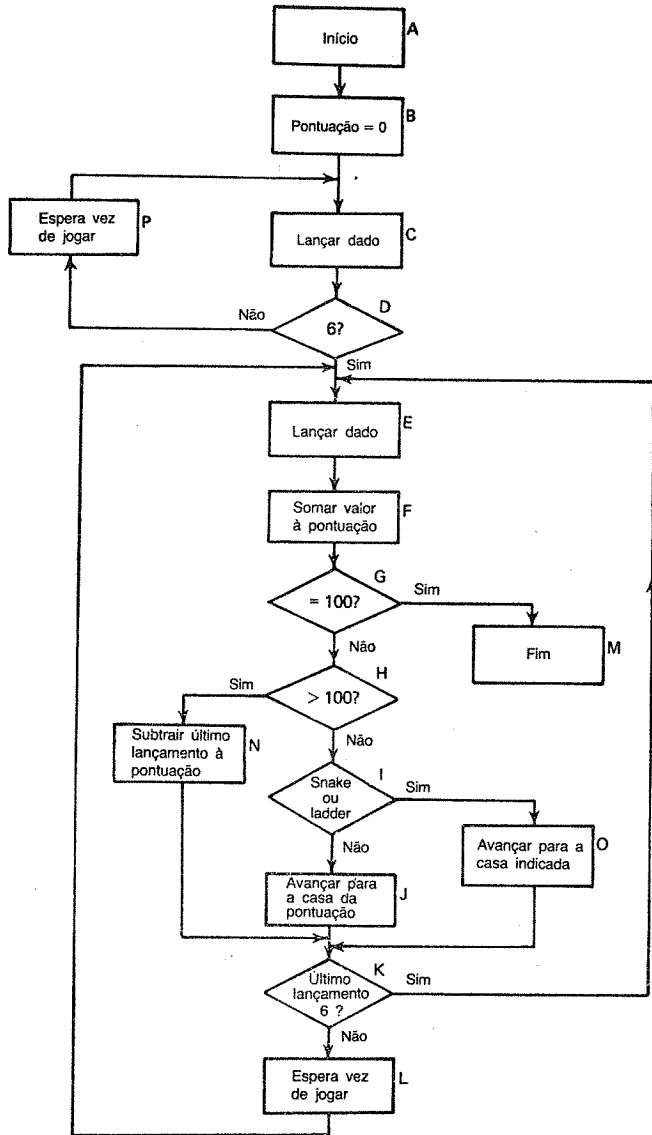



Fig. 1.1 — Fluxograma para o jogo «Snakes & Ladders».

permitirá poupar muito tempo e confusões no momento em que estiver a introduzir o seu programa no teclado ou nos interruptores do seu sistema microcomputador.

Em geral, você fará primeiramente um fluxograma genérico realizando em seguida desenvolvimentos sucessivamente detalhados desse esboço até se aproximar das

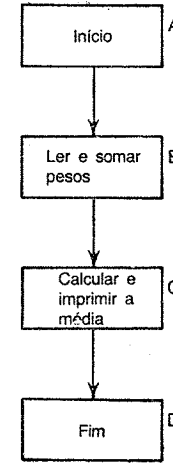



Fig. 1.2 — Esboço de fluxograma para calcular a média de vários pesos.

detalhadas ordens individuais da linguagem de programação que está a utilizar. O problema que consiste em ler o seu peso mensalmente e realizar a média anual é apresentado num fluxograma genérico na figura 1.2. Nele se representa a maneira como os humanos resolveriam o problema. Quando se começa a subdividi-lo em passos mais pormenorizados atinge-se gradualmente a maneira como o computador entende a questão.

Normalmente dispomos de um método de referenciação dos diversos níveis de um fluxograma, de tal modo que se torne possível compreender os níveis detalhados

— talvez alguns meses depois de o programa ter sido escrito — consultando apenas o seu esboço geral. Muitas vezes os blocos do primeiro nível são indicados pelas letras A, B, C, etc. No segundo nível os blocos correspondentes à fase A começam por A1 e chegam a A9; no terceiro nível vão de A10 a A99 (A1 incluindo A10 a A19, A2 incluindo A20 a A29, etc.); no quarto nível de A100 a A199 (A100-A109, A110-A119, etc.); e continuando do mesmo modo, enquanto for necessário fazê-lo. É conveniente limitar a 10 o número de subfases relacionadas com dado passo no nível anterior.

A convenção de numeração anterior é usada no desenvolvimento mais detalhado do problema dos pesos (figura 1.3). O fluxograma foi ampliado de modo a incluir a apresentação de um título e de uma mensagem indicando que todos os pesos (12) já foram comunicados à máquina. Este fluxograma introduz o importante conceito de *ciclo fechado* (loop), isto é, de uma secção do programa que será repetida muitas vezes mas que só é escrita uma vez. O bloco B do fluxograma genérico (bloco B1-B5 do mais detalhado) representa todo o ciclo neste caso. Este tipo de ciclo constitui uma técnica muito comum em programação, usada quando se sabe o número de valores que se deseja processar. O ciclo é controlado por um contador, que é (normalmente) colocado em zero antes de aquele ser iniciado. Quando se chega ao final do tratamento que se deseja aplicar a um qualquer item do ciclo, adiciona-se um ao contador, comparando em seguida o seu valor com o número de vezes total que se pretende realizar o ciclo. Se o total é inferior a este, o ciclo é novamente repetido; senão, passa-se à fase seguinte.

Para ilustrar a aplicação do ciclo fechado no problema de pesos, reduziremos o número de pesos comunicados ao computador a apenas três (se funcionar com três funcionará também com 12, ou aliás com qualquer número que se deseje), considerando vários totais. Este método de verificação de um fluxograma com um número bastante reduzido de valores constitui uma óptima técnica para avaliar a correcção de um programa.

Contagem em B1	*Total em B1*	*Peso introduzido (kg)*	*Total em B3*	*Contagem em B4*	*Resultado da comparação B5*
0	0	61	61	1	Sim
1	61	63	124	2	Sim
2	124	64	188	3	Não

Por vezes o número de vezes que pretendemos realizar o ciclo varia de cada vez que se passa o programa, pelo que não se pode fazer sempre a comparação com um dado número fixo. A maneira habitual de resolver este problema consiste em assinalar o final da entrada de dados com um número (que chamaremos «sentinela») que não pode ocorrer em quaisquer circunstâncias nas entradas válidas. Um bom no caso do problema de pesos seria 0 ou — 1, valor que mesmo a pessoa mais preocupada com a sua elegância seria incapaz de atingir. Esta técnica é muito fácil e mais flexível do que o uso de uma comparação com um número fixo. Deve-se no entanto verificar se o valor «sentinela» (que pode ser — 1) é testado *antes* de cada dado de entrada ser adicionado ao total. Se assim não acontecer, o próprio número sentinela será adicionado ao total.

O fluxograma da figura 1.4 ilustra uma solução do problema de cálculo da média de um número variável de dados terminados por um valor sentinela. O ciclo é verificado a seguir para dados de entrada de 70, 74 e 0 (sentinela).

Entrada	*Comparação em D*	*Total em F*	*Contagem depois de F*
70	Não	70	1
74	Não	144	2
0	Sim		

Os fluxogramas anteriores destroem cada dado de entrada assim que é lido o seguinte. Muitas vezes deseja-se armazenar cada uma das entradas a fim de a usar mais tarde, noutra secção do programa. Um programa para ler três dados, armazená-los e (antes de qual-

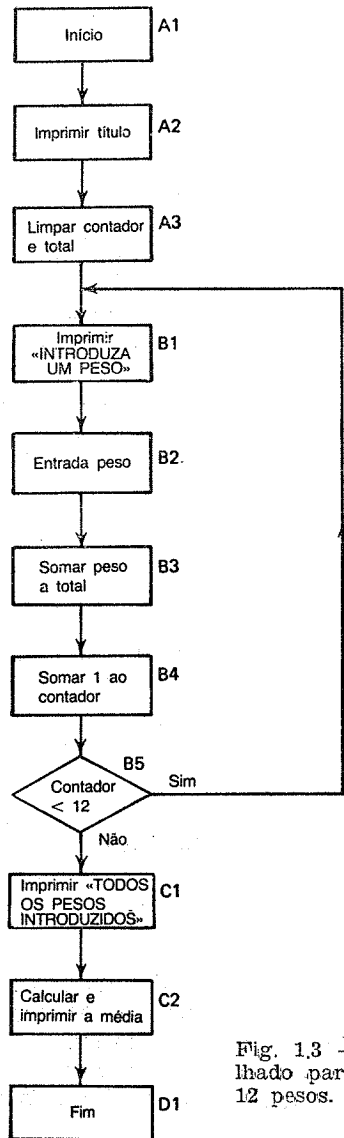



Fig. 1.3 — Fluxograma detalhado para achar a média de 12 pesos.

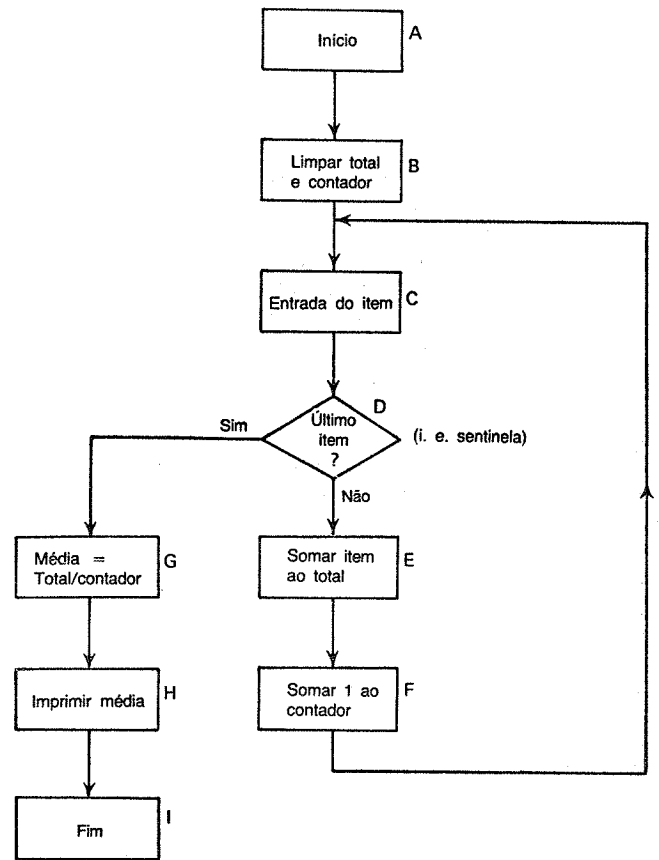



Fig. 1.4 — Fluxograma da técnica de ciclo com número «sentinela».

quer tratamento aplicado a qualquer deles) calcular a a média poderia apresentar no seu fluxograma algumas declarações como:

Ler A
Somar ao total

Ler B
Somar ao total
Ler C
Somar ao total

As repetições são óbvias neste caso, e seria impossível fazer o fluxograma se fossem tratadas 1000 entradas.

Para economizar instruções de entrada e usar a técnica do ciclo fechado nos problemas em que se deseja guardar os valores dos dados individuais, utiliza-se um conceito matemático de *quadro*, vector ou matriz («array»). Um quadro pode ser considerado uma lista de elementos individuais, cada um dos quais é identificado pela designação do quadro seguida de um índice que indica a sua posição no quadro.

Três pesos de 70, 74 e 68 kg podem ser considerados como um quadro com a designação «peso» e cujos membros ou elementos são:

$peso_1 = 70$
$peso_2 = 74$
$peso_3 = 68$,

ou numa forma mais algébrica:

$P_1 = 70$
$P_2 = 74$
$P_3 = 68$

O índice ou subscrito que indica a posição de um dado no quadro pode ser uma variável contendo um número. Se se der a K o valor 2, o elemento P_k do quadro acima terá o valor 74 (em algumas linguagens de programação considera-se que o primeiro elemento de um quadro tem um índice 0, pelo que o elemento P_2 do quadro dado como exemplo teria neste caso o valor 68).

A figura 1.5 é um fluxograma para entrada de 12 pesos, cálculo da sua média, impressão desta média e do peso menor. Os pesos individuais são guardados para

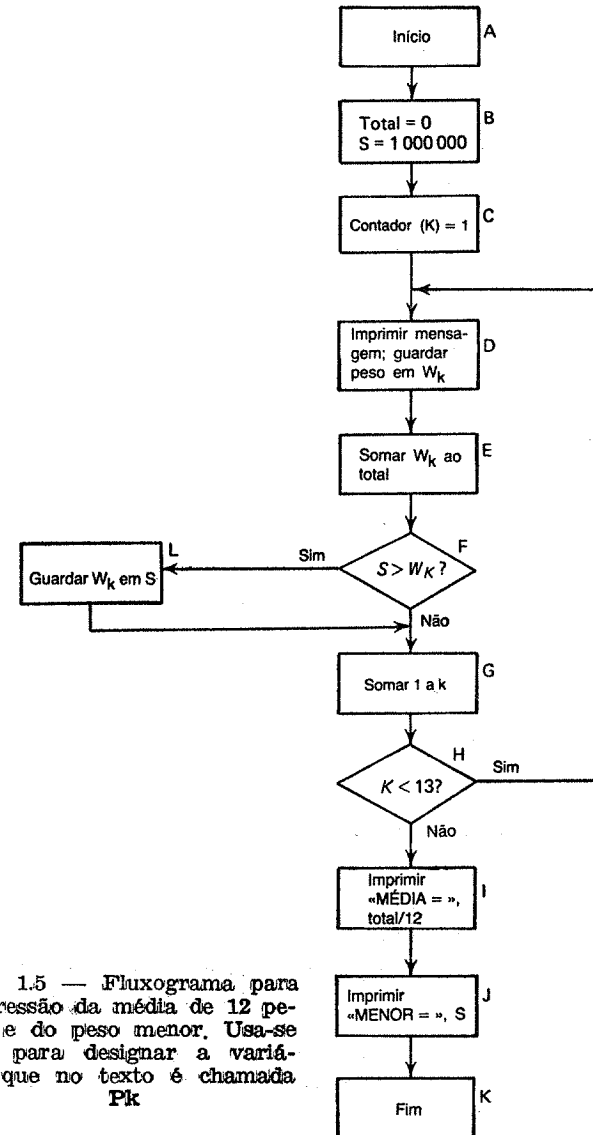



Fig. 1.5 — Fluxograma para impressão da média de 12 pesos e do peso menor. Usa-se Wk para designar a variável que no texto é chamada Pk

um qualquer uso ulterior. Neste fluxograma usa-se K para a contagem, o peso menor é armazenado em S. Para assegurar que o primeiro peso é armazenado em S quando o ciclo é executado pela primeira vez, coloca-se inicialmente um «peso mudo» de 1 000 000 em S. Dado que, em muitos quadros, o primeiro peso deve ser armazenado num elemento com o índice 1; a contagem inicia-se em 1 e em H é comparada com 13.

Se o número de pesos apresentados nas entradas é reduzido a três (de tal modo que a comparação em H se transforma em $K < 4$), certos valores transformam-se durante a realização do ciclo em:

K em D	*S em D*	*Total em D*	*Pk*	*Total em F*	*Comparação em F*	*S em G*	*K em H*	*Comparação em H*
1	1 000 000	0	70	70	Sim	70	2	Sim
2	70	70	74	144	Não	70	3	Sim
3	70	144	68	212	Sim	68	4	Não

Os valores armazenados nos três elementos do quadro P são 70, 74, 68 e o menor valor impresso na saída é 68.

Por vezes você pode pretender que os dados sejam armazenados sob a forma de pares relacionados entre si. Pode concebê-los como sendo guardados sob a forma de uma tabela com linhas e colunas (ou seja, em termos matemáticos, numa *matriz*). Cada dado de um destes pares (ou elemento de uma matriz) é indicado por dois índices: o primeiro refere-se à linha, o segundo à coluna. Nestas condições:

$A_{2,6}$ indica a sexta coluna na segunda linha.

$W_{I,J}$ indica a coluna de ordem J e a linha de ordem I; se I tiver o valor 3 e J o valor 5, indicará portanto a 5.ª coluna e a 3.ª linha da matriz W.

O fluxograma da figura 1.6 representa uma entrada de teclado com 12 pesos e alturas, para realizar um

objectivo que será definido numa secção ulterior do programa. I dá a ordem da linha e J a da coluna. No terceiro par de pesos e alturas a entrar encontram-se respectivamente 72 e 90, e portanto $P_{3,1}$ terá o valor 72.

Muitas vezes, tendemos a escrever a mesma sequência de instruções em diferentes pontos do mesmo programa ou fluxograma. Pode-se por exemplo dar entrada a diferentes tipos de dados em vários pontos do programa, pretendendo tratá-los de maneira diferente mas obtê-los no final numa tabela com o mesmo tipo de formato. Usando a lógica corrente teríamos de escrever as instruções adequadas três vezes no mesmo programa.

Os programadores são firmes defensores do princípio do mínimo esforço, e tentam extrair o máximo de cada instrução ou série de instruções. Já vimos a maneira como o ciclo fechado (loop) permite utilizar ao máximo uma série de instruções. A técnica da *subrotina* permite-nos agora escrever uma sequência muito usada uma só vez em cada programa ou fluxograma, «chamá-la» (pô-la em funcionamento) tantas vezes quantas as necessárias, e voltar sempre à instrução seguinte do programa depois da chamada à subrotina.

A figura 1.7, que não é um *verdadeiro* fluxograma, ilustra um programa no qual é dada entrada a três tipos diferentes de dados, sendo cada um deles tratado de maneira diferente dos outros. É no entanto usada uma mesma subrotina para apresentação final. As linhas espessas, contínuas e interrompidas representam os trajectos seguidos pelos três chamamentos da subrotina depois de A, B e C. Depois de cada chamamento da subrotina X a sequência do programa continua automaticamente no ponto apropriado.

A figura 1.8 apresenta um fluxograma que utiliza uma subrotina. Lêem-se dois números introduzidos por teclado, calculando-se em seguida a sua média, a média dos seus quadrados e a média dos mesmos números elevados ao cubo, imprimindo-se os resultados em seguida. O cálculo e a impressão são realizados pela subrotina «Média», chamada três vezes em C, E e G.

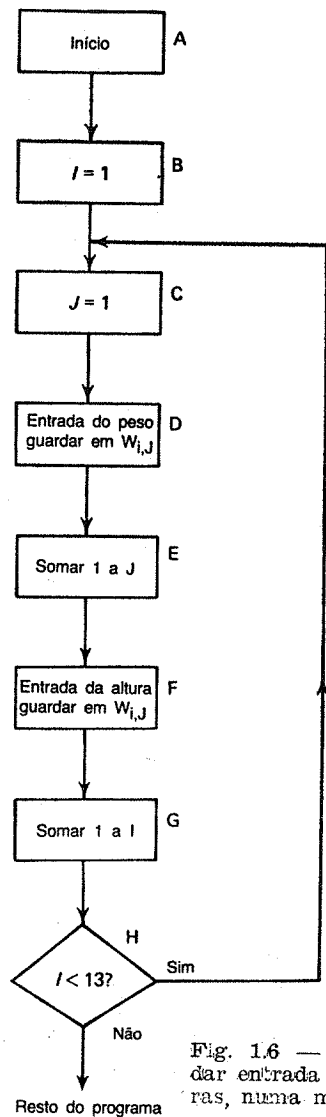



Fig. 1.6 — Fluxograma para dar entrada a 12 pesos e alturas, numa matriz.

Podem-se escrever subrotinas num programa e copiá-las em seguida para qualquer dos outros programas em que o seu uso seja conveniente. Pode-se igualmente utilizar subrotinas fornecidas pelos fabricantes ou escritas por outros. Antes de se utilizar uma subrotina que não foi escrita por nós próprios, deveremos verificar onde é necessário colocar os dados que serão utilizados por ela, como se podem passar os seus resultados para

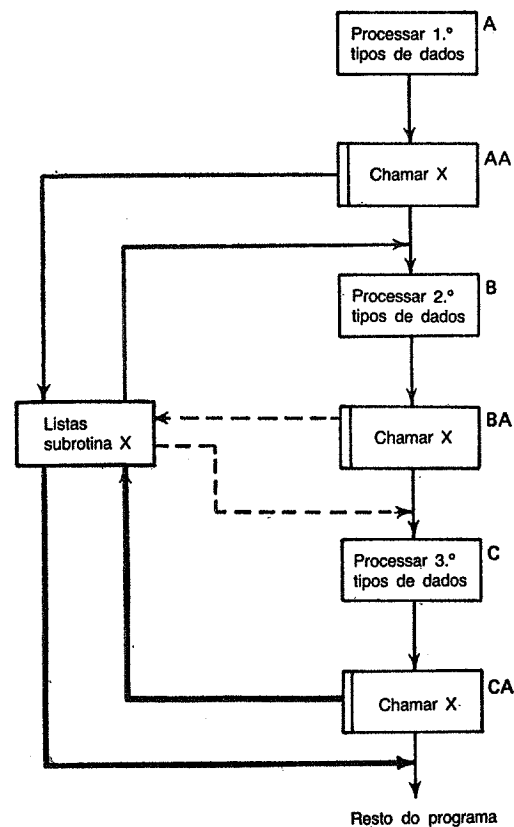



Fig. 1.7 — Chamamento e retorno de uma subrotina.

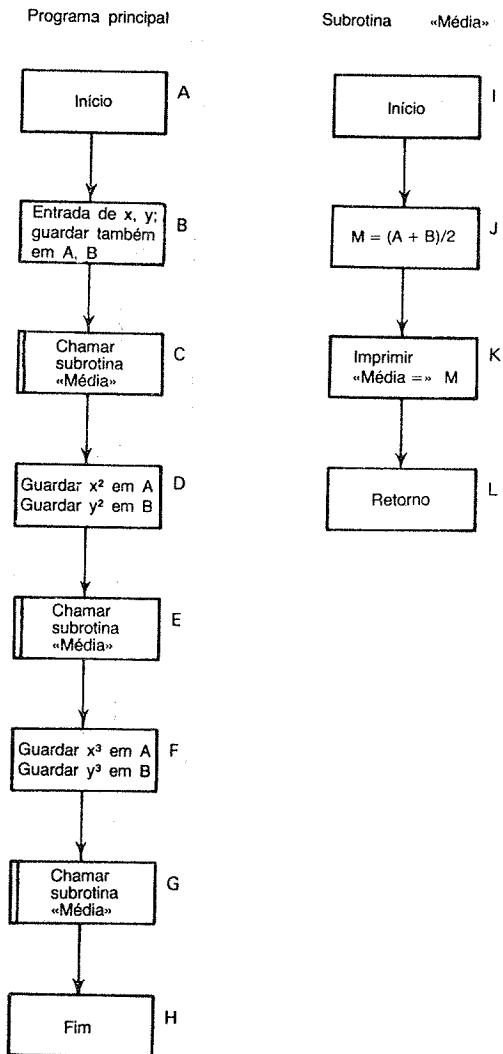



Fig. 1.8 — Fluxograma do programa principal e da subrotina.

o nosso programa, assim como quaisquer limitações de rigor da subrotina e quaisquer mensagens de erro que possa apresentar.

Têm sido feitas subrotinas para muitos fins, como cálculos matemáticos, estatísticos, científicos e comerciais, apresentação de gráficos, jogos, e até para obtenção de uma saída musical através de um gravador ou de um amplificador e colunas. Actualmente, até as máquinas de calcular programáveis são vendidas juntamente com quantidades liberais de subrotinas. Ao fazer o fluxograma, talvez seja prático marcar a instrução que introduz a subrotina com uma linha dupla, escrevendo as instruções da subrotina completamente à parte do fluxograma principal.

Para terminar este capítulo sobre fluxogramas, convém sublinhar que a construção de fluxogramas permite poupar muito tempo e esforços na codificação do programa, e ajuda-nos a detectar muitas repetições, erros e falsas premissas. Se bem que alguns profissionais de programação ponham em causa o uso de fluxogramas na forma descrita neste capítulo, o amador achá-los-á certamente muito valiosos para descobrir as deficiências lógicas do programa e ajudá-lo a transformar as suas ideias originais em linguagem de computador.

Documentação

É mais do que provável que você deseje alterar ou emendar o seu programa depois de o ter utilizado durante algum tempo. Para facilitar este trabalho, é necessário que disponha de algum tipo de documentação acompanhando o programa, dado que não é fácil extrair algum significado de um conjunto de códigos algum tempo depois de o programa ter sido feito.

Você não necessitará de arranjar para si próprio e para a sua família a documentação detalhada que o programador profissional deve apresentar, pois os programas deste são utilizados durante anos, sendo muitas vezes

acrescentados ou alterados por outros programadores. Será no entanto extremamente útil manter uma especificação do programa, descrevendo a solução e os vários níveis de fluxograma que pensa serem apropriados para o programa em causa. Verificará que estas informações são extremamente valiosas para refrescar a sua memória quando desejar alterar ou melhorar o programa original. Não necessita de escrever as informações detalhadas sobre o funcionamento do programa a que o profissional é obrigado, mas mesmo que você seja a única pessoa a usar a máquina deve guardar algumas indicações sobre a maneira como são introduzidos os dados e qual a atitude a tomar perante mensagens de erro. Essas informações constituirão um óptimo estímulo da sua memória quando voltar a usar o programa ao fim de alguns meses.

A documentação essencial que deve acompanhar o programa é normalmente escrita antes de você iniciar a codificação daquele. A descrição ou especificação do sistema e os níveis de fluxograma são tarefas essenciais que você deve realizar antes de codificar. A única documentação essencial além desta é a lista de instruções para uso do programa. Estas serão escritas durante o ensaio do programa, de tal modo que quando estiver satisfeito com a maneira como o programa funciona já esteja escrita toda a documentação. Esta será muito útil não só para outros utilizadores (quer você seja um utilizador comercial ou um amador) como, por outro lado, o ajudará a si mesmo a alterar o sistema inicial.

2

ESCOLHA DE UMA LINGUAGEM DE PROGRAMAÇÃO

À primeira vista não parece haver grandes possibilidades (nem sequer desejo) de escolher a linguagem em que os programas serão escritos. A maior parte dos sistemas microcomputadores são fornecidos com a linguagem Basic, e se é um amador que comunica instruções à máquina através de interruptores ou de um teclado hexadecimal parece não haver alternativas ao uso do código-máquina.

No entanto, uma maior familiaridade no uso do sistema e uma consulta mais prolongada da literatura de computadores, particularmente das revistas de amadores, permitir-lhe-á concluir que a maior parte dos sistemas podem ser programados com várias linguagens. O acrescento de memórias (que permitirá a um sistema pequeno utilizar a linguagem assembler e o Basic além do código-máquina, e a um sistema menos limitado de usar Fortran e Pascal para além de Basic) não é caro e existe no mercado o «software» (ou seja, os programas) necessário para usar estas linguagens.

Ser-lhe-á necessário um certo esforço para fazer a transição da escrita de afirmações no fluxograma normal para a expressão destas numa linguagem de programação. Os computadores não podem ser programados em inglês ou qualquer outra língua «natural» como o português ou o árabe. A mais simples frase tem demasiados sentidos, devido a ambiguidades gramaticais. O computador teria

dificuldade em compreender, por exemplo, que senhor é esse chamado «tempo que passa a correr»...

Os vários significados podem também ser produzidos por combinações semelhantes de letras com significados diferentes. O enorme vocabulário da linguagem natural, e o facto de existirem muitos sinónimos como:

Acrescentar 1 ao total
Adicionar 1 ao total
Aumentar o total de 1
Somar 1 ao total

para a mesma palavra ou frase, torna igualmente impossível o uso da linguagem natural para programação.

Se bem que muitos dos leitores nunca venham a usar o código-máquina e portanto não se interessem pelos dois parágrafos que se seguem, a compreensão do código-máquina e da linguagem «assembler» permitirá explorar melhor todas as potencialidades de qualquer sistema.

Código-máquina

Todos os microcomputadores possuem um conjunto de instruções incorporadas, correspondente à maneira como são construídos (ou ao seu «hardware»). Estes conjuntos de instruções diferem de uma pastilha microprocessadora para outra (se bem que a pastilha microprocessadora Zilog Z-80 obedeça a programas escritos no código-máquina da Intel 8080). Mesmo as pastilhas do mesmo fornecedor possuem diferentes códigos-máquina à medida que são acrescentadas mais instruções. O Intel 8085 tem mais duas instruções do que o Intel 8080, que por sua vez tem mais 30 instruções do que o Intel 8008, se bem que o «software» escrito para o Intel 8008 seja compatível com os dos dois últimos microprocessadores.

As instruções em código-máquina são constituídas por zeros e uns: por outras palavras, consistem em *bits* (de Binary Digits, algarismos binários) que apenas podem ter

o valor 0 ou 1. Estes bits são comunicados à máquina através dos interruptores dos sistemas microcomputadores que são programados em código-máquina. Normalmente, se um interruptor está na posição interior representa um 0, e na posição superior representa um 1. Muitas vezes, os interruptores do painel possuem luzes, que estão acesas quando representam um 1. Uma instrução típica em código-máquina será

00001000,

representando uma ordem para subtrair 1 ao conteúdo de uma memória do computador conhecida pelo nome de acumulador.

Em alguns sistemas simples (como o KIM), que possuem um teclado hexadecimal, as instruções em código-máquina são comunicadas ao computador na base hexadecimal e não em binário puro. Cada caracter hexadecimal corresponde a um grupo de quatro dígitos binários ou bits. Os números 0 a 15 em decimal, binário e hexadecimal podem ser consultados na tabela seguinte. O aspecto mais importante aqui é que os símbolos hexadecimais A a F representam combinações de dígitos binários, e *não* letras. Estes símbolos são necessários para exprimir os grupos binários 1010 a 1111 usando um único caracter para qualquer deles, e não dois, como acontece na base decimal.

Decimal (ou base 10)	*Binário (ou base 2)*	*Hexadecimal (ou base 16)*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

A instrução binária 00001000, mencionada anteriormente, poderia ser dada a conhecer à máquina através de um teclado hexadecimal sob a forma 08. O uso deste teclado torna a entrada de instruções mais rápida e mais rigorosa. Deve-se sublinhar que estes símbolos hexadecimais são traduzidos para a base dois no interior do processador, pelo que de facto a instrução obedecida se encontra de facto em código binário.

Se o seu sistema só dispõe de interruptores e luzes, você sabe certamente qual a dificuldade de comunicar à máquina as instruções necessárias para realizar qualquer programa, por muito simples que seja. Só o amador se consegue dedicar verdadeiramente à programação em código-máquina. Além do considerável tempo necessário para traduzir um fluxograma nas instruções verdadeiramente básicas do código-máquina e para depois dar entrada a esse programa no sistema, poderemos ainda considerar algumas outras desvantagens:

— A natureza detalhada das instruções; por exemplo, são necessárias três instruções para adicionar dois números e armazenar o resultado.

— O elevado número de instruções aparentemente semelhantes para o mesmo tratamento. No código do Intel 8080 existem nada menos do que sete instruções de subtracção diferentes.

— A complicação dos métodos de *endereçamento* (isto é, de indicação do local da memória do computador onde se encontra guardado o dado ou resultado pretendido).

— O uso de endereços numéricos, em vez de nomes como MEDIA) ou TOTAL, para as localizações dos dados. Isto é confuso e pode conduzir ao uso do mesmo endereço para dois dados diferentes, provocando a perda de informações vitais.

— O facto de as diferentes instruções terem comprimentos diferentes, ocupando portanto diferentes espaços na memória da máquina.

— A completa falta de semelhança com qualquer forma de comunicação matemática ou literária.

No entanto, mesmo que o seu sistema não o requeira em geral, o conhecimento da programação em código-máquina pode muitas vezes ser bastante útil. Por exemplo, se possui um sistema microcomputador programado em Basic, pode utilizar a instrução POKE (muitas vezes fornecida pelo vendedor) para introduzir ordens de código-máquina em hexadecimal, aumentando assim o programa Basic, e a instrução PEEK para examinar a correcção das instruções dadas.

Se deseja ligar o seu sistema a diversos dispositivos externos, como um sintetizador musical ou um «plotter» X-Y para construção de gráficos, será forçado a escrever algum código-máquina. Este é igualmente útil nos sistemas em «package» se não dispuser de espaço de memória dado que os programas em código-máquina utilizam menos espaço do que as instruções em Basic. Nos capítulos 8 e 9 dar-se-ão mais detalhes sobre o código-máquina e a linguagem assembler.

Linguagem assembler

O conceito de substituição de uma lista de caracteres por outra, que constitui a base fundamental da tradução de línguas, foi rapidamente aplicado à simplificação do processo de programação.

Uma linguagem «assembler» utiliza uma representação mais compreensível das duas partes de uma instrução em código-máquina: o código de *operação* (que indica adição, comparação, etc.) e o *operando*, que indica o ende-

reço dos dados a que a operação é aplicada. Um programa de tradução, normalmente designado *assembler*, realiza a tradução da linguagem, a que se dá o mesmo nome, para as instruções binárias a que a máquina de facto obedece. Os códigos de operação têm normalmente uma forma mnemónica como ADD (somar), SUB (subtrair).

Se você não dispõe de um teclado de máququina de escrever ligado ao sistema microcomputador, que é essemcial para trabalhar em linguagem assembler, pode no entanto utilizar ainda esta linguagem para escrever o seu programa, traduzindo-o em seguida, instrução a instrução, à medida que o comunica à máquina através de interruptores ou do teclado hexadecimal. Se bem que a linguagem assembler tenha muitos dos defeitos do código-máquina (a saber, instruções muito detalhadas, incapacidade para ser utilizada nos computadores de outros fabricantes, muitas instruções quase idênticas), e só deva ser usada na prática quando se é forçado a fazê-lo ou quando se pretende escrever um programa que ocupe pouquíssimo espaço, permitirá de facto conseguir uma maior velocidade de trabalho e um maior rigor do que o código-máquina.

Uma declaração comum em linguagem assembler tem três partes:

— um *rótulo*, nome que é escolhido para o endereço em que é colocada a instrução;

— o *código de operação*, que assume uma forma mnemónica indicada pelo fabricante;

— o *operando*, que especifica o registo, endereço ou valor literal dos dados a que é aplicada a operação.

Normalmente estas três partes são seguidas de um comentário, que descreve o objectivo da instrução e pode referir-se a um «bloco» do fluxograma. O campo do rótulo é opcional, e só é de facto usado quando se pretende indicar a localização de uma dada instrução contida noutra parte do programa. Por exemplo, se pretende voltar a um dado ponto de um programa e se o *acumulador*

(definido no capítulo 8) não é zero, pode-se rotular a instrução a que se pretende voltar TOTAL (ou qualquer outro nome por nós escolhido); então, na instrução de retorno escreve-se:

JNZ TOTAL

Vejamos algumas instruções típicas em linguagem assembler:

START	LDA ITEM	Colocar um «item» de um endereço no acumulador (LDA — «Load», carregar).
	ADD X	Somar um item a que chama X ao conteúdo do acumulador (ADD — somar)
	MOV C,A	Deslocar o conteúdo do acumulador para o registo C (MOV — «Move», deslocar)
	JMP LOOP	Saltar para o rótulo LOOP («ciclo»), e continuar a obedecer sequencialmente as instruções que se encontram desse ponto para diante (JMP — «Jump», saltar)
	CMP M	Comparar o conteúdo do registo a que chamamos M com o do acumulador (CMP — «Compare», comparar)
CLOSE	HLT	Parar (guardado num endereço a que chamamos CLOSE) (HLT — «Halt», parar).

Nos capítulos 8 e 9 apresentam-se muitos outros exemplos de declarações em linguagem assembler.

Apesar de não constituir um meio verdadeiramente apropriado à escrita de programas científicos e técnicos ou para o tratamento de aplicações comerciais, muitas instalações de minicomputadores utilizam a linguagem assembler. Ainda mais surpreendente é o número de computadores grandes que ainda utilizam esta linguagem

como método principal de programação, a fim de economizar a memória do computador.

Linguagens de alto nível

Alguns leitores recomeçarão o seu estudo neste ponto, dado que foi sugerido mais atrás que omitissem os dois parágrafos anteriores se não vissem interesse em consultar um breve resumo da natureza e potencialidades do código-máquina e da linguagem assembler. Alguns ficarão perturbados com a aparente dificuldade que parece ser presente, mesmo sob a forma de um breve resumo, à programação em código-máquina ou em linguagem assembler. Como já se disse, deve-se tentar evitar estes tipos de linguagem de programação a menos que não se tenha outra alternativa ou se goste de trabalhar tão próximo do «hardware» de um dado processador.

A maior parte dos programas usados nas instalações de computadores comerciais, científicos, técnicos e educativos são escritos numa linguagem de alto nível. A maior parte dos microcomputadores em «package» para amadores permitem igualmente o uso de uma destas linguagens, em geral a designada por Basic.

Uma linguagem de alto nível utiliza termos que nos são familiares, em vez de um código difícil que é mais apropriado ao «hardware» de um dado sistema microcomputador. Como resultado, é necessária muita tradução para converter uma única declaração em linguagem de alto nível para a linguagem que o computador compreende. Se o seu sistema permite usar o Basic, tente escrever uma instrução como:

```
PRINT (9 + 56)/(6 + 9 — 2) + 300
```

e carregue na tecla RUN; a máquina apresentará o resultado 305. Se tiver estudado o parágrafo anterior sobre código-máquina e linguagem assembler compreenderá a quantidade de trabalho realizado pelo sistema para receber a instrução anterior e traduzi-la, tendo particularmente em vista o facto de serem necessárias complicadas instruções de transporte do dígito mais significativo quando um resultado aritmético é superior a 255 (o maior número que pode ser guardado em oito bits: 1111 1111) e que a máquina não possui a operação divisão. Algumas outras declarações em linguagem de alto nível serão:

```
ADD BONUS TO PAY GIVING GROSS-PAY
IF COUNT = 12 THEN GO TO 100
a = (c/log(d — e) + (if y = z then h else sqrt (v — Z))
```

Os programas que traduzem as instruções de alto nível são conhecidos pelo nome de *compiladores* ou *interpretadores*. Estes programas compiladores constituem a forma vulgar de tradução no caso dos computadores grandes e dos minicomputadores. Um compilador é um programa que aceita todo o programa em linguagem de alto nível e o traduz no seu conjunto. Um interpretador traduz cada instrução de programa à medida que a recebe, por exemplo no problema de pesos do capítulo 1, cada instrução do ciclo deveria ser traduzida 12 vezes. Em alguns sistemas microcomputadores pode-se escolher entre o uso de um interpretador Basic ou um compilador Basic. Um compilador pode necessitar de mais espaço de memória do que um interpretador, mas por outro lado não se encontra na memória quando o programa por ele traduzido está a ser obedecido.

É necessário ter em conta o volume de alguns programas interpretadores se o seu sistema possui uma memória bastante reduzida, dado que aquele volume impedirá a escrita de programas de grandes dimensões. Uma outra desvantagem é a velocidade relativamente lenta de execução do programa dado que é necessário traduzir cada instrução assim que se pretende aplicá-la. O programa traduzido ou *programa-objecto* produzido por um compilador é compacto, mas o compilador é menos prático do que um interpretador quando se pretende introduzir pe-

quenas alterações num programa, dado que estas envolvem sempre um elemento de recompilação. Como terá descoberto se tiver experimentado a instrução PRINT acima indicada, é mais fácil experimentar quando se usa um programa interpretador. É mais fácil introduzir as alterações. A maior parte dos sistemas microcomputadores em «package» para amadores são acompanhados por um programa interpretador, que se encontra incorporado na máquina ou é lido para a memória da máquina a partir de uma cassette.

Existem mais de mil linguagens de computador, a maior parte das quais são obsoletas, raramente usadas, apenas usadas em trabalhos específicos como animação e controlo de máquinas-ferramentas, ou não foram implementadas nos microcomputadores. Estão constantemente a ser desenvolvidas novas linguagens, particularmente nas instituições académicas, numa tentativa de facilitar cada vez mais o problema da comunicação com o computador; mas a maior parte destas raramente são usadas fora do meio onde foram concebidas.

Algumas das características de uma linguagem de programação ideal são as seguintes:

1. Independência relativamente ao hardware específico do computador, pelo que o programa pode ser utilizado sem alterações numa grande variedade de computadores, incluindo, sistemas microcomputadores. Isto raramente é conseguido na prática.
2. Definição rígida, de modo a que aqueles que escrevem programas compiladores não façam interpretações diferentes das várias características da linguagem. Isto ajuda a promover a universalidade atrás mencionada.
3. Seja normalizada por alguma instituição nacional (normalmente a ANSI, ou seja, a American National Standards Institution).
4. Utilize termos familiares para o utilizador, e seja adequada à resolução dos seus problemas sem necessidade de recorrer ao código-máquina ou à linguagem assembler para cobrir as suas próprias deficiências.
5. Os símbolos usados devem ser familiares e estarem presentes no teclado normalmente usado na maior parte dos computadores.
6. Deve ser fácil de aprender e utilizar.

Poucas ou nenhumas linguagens apresentam todas estas características. É conveniente não perder muito tempo a definir a linguagem «ideal» para uma dada aplicação. Todas as comumente empregues têm os seus defensores. Segue-se a descrição resumida das linguagens de alto nível mais vulgares.

Basic

Esta linguagem tem a vantagem de ser usada na maioria dos microcomputadores. Foi primeiramente lançada em 1965, tendo-se verificado entretanto que é bastante fácil de usar — particularmente por programadores não profissionais. Existem muitos programas nesta linguagem, que podem ser transferidos com poucas alterações para qualquer sistema microcomputador. Todos os números das revistas de computadores para amadores contêm invariavelmente um programa Basic que pode ser adaptado ao seu sistema.

As suas principais desvantagens são:

1. Falta de normalização. Existe um esboço de normalização para uma linguagem Basic mínima que está a ser estudada pela American National Standards Institution, mas no momento em que este livro é escrito trata-se ainda de uma *proposta* de normalização. A ANSI está a trabalhar também na normalização de uma linguagem Basic ampliada. No entanto existe uma base comum a todas as variantes Basic, que é descrita nos capítulos 3 a 6 deste livro, e a que cada fabricante acrescenta variantes e potencialidades extra. O capítulo 7 dá

a conhecer algumas das variantes do Basic, vulgarmente encontradas nos sistemas microcomputadores.

2. A base comum citada não possui potencialidade de tratamento de ficheiros, e os seus formatos de saída e métodos de tratamento de grandes quantidades de dados são primitivos.

3. Não pode normalmente usar nomes como PAY e STOCK para os dados.

Cobol

Esta linguagem é a mais vulgar para programas comerciais usados em computadores grandes, sendo concebida para tratamento de ficheiros comerciais. Permite ao utilizador usar nomes mnemónicos para os seus dados, como TAX e PAY. Foi normalizada duas vezes, nas normas ANSI 68 e ANSI 74. Existem poucos compiladores desta linguagem para microcomputadores. Alguns dos que existem são compiladores *cruzados*, nos quais é necessário compilar primeiramente o programa num minicomputador ou num computador de grandes dimensões, só depois sendo possível transferi-lo para o sistema microcomputador.

O Cobol é superior ao Basic para tratamento de dados comerciais, existindo um elevado número de profissionais que realizam programação nesta linguagem. É imprópria para trabalho matemático; a escrita de um problema técnico em Cobol já foi comparada à tentativa de tirar as cascas a ervilhas com luvas de boxe... É improvável que encontre um compilador Cobol para um sistema pequeno. Muitos computadores usados em pequeno comércio têm no entanto verificado que conseguem resolver sem grande dificuldade as limitações do Basic no tratamento de dados comerciais.

Fortran

Trata-se de uma linguagem matemática na qual têm sido escritos mais programas do que em qualquer outra.

Foi normalizada pela ANSI, se bem que muitos programas tivessem sido escritos antes desta normalização. Existem provavelmente mais programadores profissionais em Fortran do que em Basic, dado que esta linguagem existe desde os anos 50. Os seus requisitos de memória para um compilador (não é provável que encontre um programa interpretador) são grandes, e foi concebida principalmente para tratamento de problemas numéricos e não para uso «conversacional» através de um teclado de máquina de escrever. Pode-se chamar às variáveis os seus nomes habituais, como acontece em Cobol, e as possibilidades de subrotinas são de longe superiores às do Basic, que aliás derivou do Fortan.

Pascal

Uma linguagem relativamente nova para trabalho matemático, ainda não foi aplicada em microcomputadores e não é apoiada pelo grande número de programas publicados que existem para a Fortran. Um programa interpretador pode necessitar de muito espaço de memória. Está definida de maneira rígida, mas ainda não foi normalizada por qualquer instituição. Combina as características matemáticas do Fortran ou do Basic com a capacidade para realizar tratamento de ficheiros. Infelizmente, é necessário declarar todos os nomes antes de programar, pelo que não é possível escrever no teclado

```
LET A = B/9
```

como em Basic sem declarar se A e B são variáveis inteiras, caracteres, variáveis decimais, matrizes, etc. Não existe um operador para exponenciação, mas dispõe de funções que algumas linguagens matemáticas não possuem.

Algol

Existem duas versões, o Algol 60 e o Algol 68, ambas rigidamente definidas mas incompatíveis entre si. É uma

linguagem intelectualmente satisfatória para problemas matemáticos e favorecida em círculos académicos europeus, mas até ao momento em que este livro é escrito ainda não foi aplicada aos sistemas microcomputadores.

PL/I

Trata-se de uma linguagem de uso geral originalmente concebida para os computadores IBM mas mais tarde estandardizada e aplicada a outros computadores de grandes dimensões. É igualmente satisfatória para trabalho comercial e matemático. As linguagens de programação de sistemas PL/M, PL/6800 e PL/Z para os microprocessadores Intel, Motorola e Zilog devem alguma coisa à original linguagem PL/I.

APL

Trata-se de uma linguagem bastante completa, com potencialidades de tratamento de números e de caracteres de texto. Infelizmente utiliza um conjunto de caracteres bastante fora do normal, sendo como consequência bastante pouco usada nos sistemas microprocessadores, apesar de terem sido introduzidos no comércio programas interpretadores para ela. Talvez deva ser considerada como a linguagem de alto nível mais avançada e completa das actualmente existentes. Permite a utilização de matrizes, sendo muito útil para o tratamento de blocos de caracteres de texto.

3

INTRODUÇÃO AO BASIC

Entrada de programas

Os programas Basic dão entrada no computador através de um teclado ligado a um écran video (VDU) ou a uma tele-impressora. Antes do mais é necessário ao leitor habituar-se à disposição do teclado. O tipo mais vulgar consiste no vulgar teclado «QWERTY» com um conjunto *completo* de números 0 a 9 na fila superior. Os operadores experientes notam que a maior parte das versões de Basic *não* aceitam o «l» e o «O» alfabéticos como tratando-se dos números 1 e 0. Alguns computadores domésticos, como o Pet e o Sol, possuem um teclado semelhante ao de uma máquina de calcular à esquerda do teclado alfabético. Isto reduz as possibilidades de confusão na entrada de números.

Além das teclas normais de uma máquina de escrever, encontram-se nestes teclados alguns caracteres especiais. Parte deles, como os gráficos Pet, são únicos, sendo empregues apenas para um dado computador e são convenientemente explicados no manual do fabricante. Outros, como «Line feed» (alimentação de linhas), «Return» (retorno), e «Rubout» (que elimina uma linha completa) são bastante vulgares.

Os caracteres apresentados na metade superior de cada tecla, e que são obtidos premindo uma tecla especial (como nas máquinas de escrever normais para obter as maiúsculas) devem ser cuidadosamente estudados. Um

dos mais vulgares em Basic, '=', encontra-se na parte superior na versão mais comum de teclado. A maneira mais vulgar de eliminar um caracter mal batido consiste em premir na tecla de voltar atrás, como nas máquinas de escrever. Esta encontra-se normalmente marcada com uma seta apontada para trás.

Se o seu teclado está ligado a um VDU, você verificará que é bastante útil dominar tão depressa quanto possível o uso do *cursor*, o pequeno traço de luz no écran, que indica a posição em que irá surgir o carácter batido em seguida. No teclado encontrará sempre teclas que controlam o movimento vertical e horizontal do cursor. Não destrói qualquer carácter sobre o qual passe. Quando o cursor é colocado sobre um dado carácter, pode ser inserido outro novo em substituição do anterior. A menos que esteja a trabalhar em «edit mode» (modo de edição), a linha deve ser corrigida por manipulação do cursor antes de dar entrada na máquina quando se toca na tecla de retorno.

É conveniente não introduzir mais caracteres em cada linha do que as recomendadas no manual, dado que algumas versões de Basic não aceitariam a linha.

Existem grandes variações, nas diversas versões de Basic, quanto ao tratamento dos espaços não significativos de uma linha. O manual do computador com que trabalha, ou da versão de Basic que implementa num computador por si montado, devem ser sempre consultados quanto a este ponto. Normalmente, quando é permitido ou exigido um espaço, é possível introduzir vários. Este livro indicará as várias instruções em que os espaços assumem grande importância.

Provavelmente verificará que está a cometer erros na entrada inicial dos programas. No entanto, nenhum programa é obedecido pela máquina até você tocar a tecla RUN, pelo que os erros de digitação das linhas de modo algum podem deteriorar o equipamento. Se só está familiarizado com as máquinas de escrever não eléctricas achará as teclas muito sensíveis; bastará uma pequena pressão para dar entrada a um carácter, sendo muito

comum uma «entrada dupla» no caso dos principiantes. Se pretende dar entrada para uma cassette a programas longos ou grandes ficheiros de dados (que devem ser tratados através do teclado a menos que tenha dinheiro para investir num leitor de fita perfurada ou de cartões perfurados), é bom que se habitue a escrever bem neste teclado. Oito dedos são invariavelmente mais eficazes (e depois da aprendizagem inicial mais rigorosos também) do que um. Depois de ter aprendido a maneira de programar cadeias de caracteres, no Capítulo 6, será capaz de escrever programas para verificar o rigor da sua digitação e testes de concepção para facilitar a correcção de quaisquer erros repetidos.

Inicialmente, pode-se treinar a entrada de linhas segundo o «modo calculadora» do Basic, que reduz o computador ao nível de uma máquina de calcular mas é útil para nos familiarizarmos com a entrada de dados. Se escrever:

```
PRINT 11 + 6
```

e em seguida bater a tecla «Return», o computador imprimirá:

```
17
```

Você pode usar quaisquer sinais aritméticos e familiarizar-se com o sinal «*» usado para multiplicar e o sinal «/» usado para dividir. Pode realizar mais do que um cálculo na linha única que pode usar em «modo calculadora», pelo que se escrever:

```
PRINT 29 — 10, 6/2
```

o computador imprimirá:

```
19          3
```

(O termo «print» será usado tanto para a saída impressa como para a saída realizada através de um VDU).

Comandos do sistema

Até agora ainda não fizemos nada no teclado que não pudesse ser feito com uma máquina de calcular barata. Antes de escrever programas, no entanto, convém examinar parte das ordens dadas ao computador (comandos) que não constituem instruções do programa mas sim instruções ao interpretador, compilador ou sistema de operação (termos já explicados no capítulo 2), levando-os a executar certas funções com o seu programa. Encontram-se diversas variantes, mas os mais vulgares são:

RUN NEW LIST LOAD SAVE

O Basic usado nos terminais de muitos minicomputadores e grandes computadores possui vários outros comandos do sistema como CATALOG, UNSAVE, RENAME, etc. Estes não se encontram porém nas implementações de Basic nos microcomputadores. O significado dos comandos citados será explicado a seguir.

RUN

Este é o comando mais vulgar. Em alguns equipamentos existe uma tecla «RUN» para eliminar a necessidade de escrever esta palavra totalmente. Quando o comando dá entrada, o programa que se encontra em memória é executado, pelo que o comando só deve ser dado quando se está certo de que o programa em causa está livre de erros. Quando se trabalha no «modo calculadora», caso em que só é executada uma linha de cada vez, premir a tecla «Return» equivale a premir a «Run». Se tocar na tecla RUN segunda vez depois de o programa ter sido executado, o computador repeti-lo-á.

NEW

Este comando limpa o programa que se encontra em memória e permite dar entrada a um novo programa.

Todos os valores entrados para o programa que se encontrava na memória antes de se tocar a tecla NEW são eliminados. Em muitos sistemas há o perigo de, se não se bater NEW, o novo programa ser adicionado ao anterior.

LIST

Este comando permite listar o programa em memória. É útil quando se realizam bastantes alterações ou, no caso de um VDU, para remover do écran as últimas instruções e observar novamente o início do programa.

LOAD

Este comando carrega um dado programa existente numa cassette ou num disco para a memória da máquina. Se for seguido por uma designação de programa, por exemplo

LOAD «ANNE'S BUDGET»

procurará esse programa e carregá-lo-á na memória, pronto para ser executado.

SAVE

Este comando escreve o programa existente em memória numa cassette ou num disco. Se deseja recuperar o programa indicando o seu nome, deve seguir a palavra SAVE por esse nome.

Números de linha

Se está a escrever um programa com mais de uma linha, cada declaração deve ter associado um dado número de linha. Esta é outra diferença entre as instruções de

programa e os comandos do sistema, por exemplo LIST: estes últimos não se encontram associados a números de linha.

Os números de linha começam por 1. O limite superior varia para cada sistema. Não é provavelmente prudente utilizar números de linha superiores a 9999. É habitual deixar pelo menos um espaço entre o número de linha e a instrução, mas não convém deixar qualquer espaço no interior do número propriamente dito. Por exemplo:

```
10 PRINT 7*5
```

estará correcto, mas

```
80 0 PRINT 66/3
```

pode provocar problemas.

É habitual deixar intervalos de 10 entre os números de linha a fim de possibilitar a inserção de declarações. A última linha de um programa é a declaração END, que por convenção tem muitas vezes o número 9999. Um programa Basic muito simples poderia ter a seguinte aparência:

```
10   PRINT 16/8
20   PRINT .5*6
30   PRINT 3 + 1
40   PRINT 52 — 47
9999 END
```

A sequência completa de comandos e declarações para fazer passar este programa seria:

```
NEW
10   PRINT 16/8
20   PRINT .5*6
30   PRINT 3 + 1
40   PRINT 52 — 47
9999 END
RUN
```


Este programa produziria a saída:

```
2
3
4
5
```

Normalmente o computador apresenta na saída as linhas por ordem, pelo que a inserção de uma nova versão de uma dada linha antes da declaração END provocaria a substituição da linha que anteriormente existia no programa com o mesmo número de linha. Pode-se introduzir pelo teclado em qualquer ponto do programa uma linha omitida pois esta será obedecida pela sequência correcta. Por exemplo:

```
10   PRINT 8*12.5
20   PRINT 900/3
30   PRINT 780 — 380
15   PRINT 140 + 60
9999 END
```

seria obedecido pela ordem correcta de linhas, dando na saída:

```
100
200
300
400
```

Comentários

A declaração REM introduz um comentário que não é compilado ou interpretado de modo a produzir uma declaração. É útil para dar título ao programa ou para explicação das declarações. Pode preceder linhas em branco para espaçar secções diferentes do programa, e linhas de «orna-

mento» para melhoramento da apresentação do texto, por exemplo:

```
10 REM PROGRAMA ESCRITO POR ANNE PATRICIA
20 REM 19 MAY 1978
30 REM ++++++++++++++
40 REM FAZ HOROSCOPOS
50 REM E CHAMA-SE
60 REM ************************
70 REM NOSTRADAMUS
```

Por razões de espaço, REM não será uma declaração muito usada nos exemplos apresentados neste livro. A sua importância, no entanto, nunca será demasiado sublinhada, em particular se os seus programas forem usados por outras pessoas. Se (como o autor espera), você tirar algum lucro do seu «hobby» e vender os seus programas, as anotações apropriadas torná-los-ão mais vendáveis.

A declaração LET

```
10 LET A = 3
```

O tipo declaração acima indicado é o mais vulgar em Basic. Coloca o valor que se encontra à direita da declaração (3) na parte da memória do computador que tem a designação apresentada à esquerda (A). Permite portanto guardar valores processados, pelo que a seguinte sequência de declarações:

```
20 LET X = 7*5
30 LET D = 19 — 8
40 LET Q = X — D
50 PRINT Q
```

daria na saída:

```
24
```


As designações dos «blocos» de memória do computador usados para guardar informações são conhecidas pelo nome de *variáveis*. O conteúdo anterior da variável do lado esquerdo de uma declaração LET é substituído pelo valor apresentado do lado direito desta declaração.

O lado direito da declaração LET pode ser:

— Um número;
— Uma outra variável;
— Uma «fórmula» ou expressão aritmética, que pode consistir em variáveis, números, ou ambos.

Em seguida apresentamos exemplos dos três tipos de declaração LET:

```
60 LET V = 10
70 LET P = A
80 LET R = Y/4 — T
```

Em muitas versões de Basic a palavra «LET» pode ser omitida. Se, no entanto, você pretende que o seu programa trabalhe na maior parte das máquinas, é conveniente nunca esquecer a palavra «LET».

Nomes das variáveis

Estes nomes são normalmente restringidos às 26 letras do alfabeto inglês ou a qualquer letra seguida de um único algarismo; podem ser empregues portanto 286 variáveis. Convém marcar no seu fluxograma a utilidade das variáveis de um dado programa, por exemplo que A contém pagamentos e B impostos. A declaração REM pode igualmente ser útil para o recordar do conteúdo das variáveis e ajudá-lo a evitar a destruição do valor de uma variável que contém informações importantes. Convém ainda evitar as letras O, S e Z, que (em particular quando escritas à mão) podem ser confundidas com algarismos.

Números

Os números ou *constantes numéricas* são usados para inserir um dado valor numérico numa variável. Podem ser usados por si mesmos (como acontece nos exemplos de «modo calculadora» dados anteriormente neste mesmo capítulo) ou numa combinação de variáveis, por exemplo:

```
90 LET U = 3.6
100 LET K = U — 2
```

Existem três maneiras de dar entrada aos números.

a) *Números inteiros.* Estes podem ter ou não sinal, pelo que 67 e + 67 são igualmente válidos. Não se permite o uso de vírgulas a separar os milhares, milhões, etc. (os programas para computadores utilizam como se sabe não só instruções mnemónicas em inglês como ainda a notação numérica usada nesta mesma língua; a vírgula equivale portanto ao ponto que se usa em português para indicar milhares, enquanto o ponto da notação inglesa equivale à vírgula usada em português para indicar a parte decimal do número); por exemplo, 65,000 não será aceite pela máquina. Os zeros à esquerda (por exemplo 0016) são ignorados. Algumas versões pequenas do Basic apenas trabalham com números inteiros.

b) *Números decimais.* A parte inteira de um número pode ser omitida numa fracção decimal. Exemplos de números válidos para apresentação à máquina:

1.96
— 11.640
+ 5.3
0.23
.02877

c) *Forma exponencial.* Esta só tem interesse para o matemático ou cientista que deseje dar entrada na máquina a números muito grandes ou muito pequenos, que

dificilmente poderiam ser comunicados a esta numa das duas formas já descritas. A forma exponencial é a seguinte: mantissa, letra E, expoente (potência de 10). Assim: 53.7E — 2 representa 0,537. Esta forma de representação é usada em máquinas de calcular para cálculos científicos.

Pode-se introduzir sinais em cada uma das partes do número. Os zeros à esquerda são ignorados. O número 567 pode portanto ser representado em diversas formas exponenciais, como:

56.7E+1
+5.67E2
.567E03
567E0
5670E — 1
0.000000567E9

É melhor evitar bater espaços quando se insere um número em formato E.

Todos os computadores têm um limite quanto à dimensão dos números que podem ser representados (o maior número que o Pet pode conter é 1.70141183E38). Surgirá uma mensagem de erro sempre que uma constante numérica ou o resultado de um cálculo estiver fora da gama de números representáveis. Por vezes, no entanto, quando um número é muito pequeno e ocorre a condição designada normalmente por «overflow», é dado o resultado 0 e o programa continua, como acontece no Pet. Deve-se consultar o manual do sistema para conhecer a gama de valores que o computador pode tratar. Esta gama não constitui normalmente uma limitação — mesmo o número de átomos presentes em 1,008 gramas de hidrogénio (606E21) pode ser usado sem dificuldade na maior parte das implementações de Basic em microcomputadores.

No caso dos leitores que tratem números relativamente grandes ou pequenos, convirá examinar a precisão com que os resultados são dados no microcomputador usado. Normalmente as respostas são rigorosas até seis

ou sete algarismos significativos (esta definição é um tanto vaga, dado que os computadores trabalham na base 2 e não na base decimal).

Operadores aritméticos

Já encontrámos até aqui os quatro operadores aritméticos:

+ — * /

Existe ainda um quinto:

↑ Elevar a uma potência (exponenciar)

Este operador não é porém muito usado, e portanto é omitido em algumas versões mais pequenas de Basic como a Tandy Level I Basic e a Computer Workshop 3 K e 4 K Basic. Quando se eleva um número a uma potência inteira é sempre mais eficaz utilizar multiplicações repetidas. Por exemplo:

```
110 LET P = A*A*A
```

é mais eficaz em termos de tempo de tratamento do que:

```
110 LET P = A↑3
```

Calculou-se que a elevação ao quadrado é realizada 30 por cento mais depressa e ao cubo 15 por cento mais depressa pelo primeiro método, pelo que em certo sentido não se está a perder muito quando se utiliza uma versão de Basic que não possua o operador ↑.

A expressão 0 ↑ 0 é considerada como igual a 1. Zero elevado a qualquer outra potência é igual a zero. Os números podem ser elevados a potências fraccionárias e negativas utilizando o operador ↑:

```
120 LET V = 25↑(1/2)
```

daria o valor da raiz quadrada de 25, ou seja 5, à variável V.

Combinação de operações aritméticas (expressões aritméticas)

As variáveis e os números podem ser combinados em expressões como:

```
130 LET L = 3*L + 1
```

À primeira vista a expressão acima parece constituir um contra-senso aritmético. No entanto, o « = » numa declaração LET *não* significa igualdade, servindo pelo contrário para separar a variável a que será atribuído o valor da expressão que deve ser calculada para obter este valor. No exemplo citado, é substituído o valor anteriormente dado a L. Uma forma bastante vulgar de declaração Basic para incrementar contadores é:

```
140 LET K = K + 1
```

São possíveis combinações complexas de operações formando expressões, por exemplo:

```
150 LET Q = 48 — P ↑ 3 + F
```
$(q = 48 - p^3 + f)$

mas é melhor evitá-las se tem quaisquer dúvidas quanto à maneira como o Basic calcula as expressões. Vamos indicar em seguida algumas regras gerais quanto à formação das expressões aritméticas em Basic:

1. Os operadores nunca são implícitos, pelo que 2AB é «ilegal» em Basic e deve ser escrito 2*A*B.

2. Um sinal menos é tratado como uma subtracção e não como um sinal negativo; por exemplo, — 4 ↑ 2 será considerado pela máquina como $-(4^2)$, ou seja — 16, e não como $(-4)^2$, ou seja, + 16.

3. Dois sinais ou operadores nunca se devem encontrar a seguir um ao outro, pelo que Y* — 3 é também uma expressão ilegal. Usam-se parêntesis para vencer esta

dificuldade, devendo a expressão anterior ser escrita do seguinte modo: Y* (— 3).

4. Qualquer expressão pode ser colocada entre parêntesis. Se deseja escrever expressões aritméticas complexas, utilize parêntesis sempre que tem dúvidas sobre a maneira como a máquina interpreta as indicações que lhe fornece. Os parêntesis redundantes são ignorados pelo computador, desde que se encontrem aos pares. Uma expressão entre parêntesis é tratada como uma variável ou número únicos, desde que a expressão em causa não se encontre imediatamente a seguir a outra expressão entre parêntesis, a uma variável ou a um número sem sinal. Por exemplo:

160 LET V = 7(X + Y) é incorrecto;
160 LET V = 7*(X + Y) é correcto.

Os parêntesis ajudam a resolver quaisquer ambiguidades quanto à ordem de avaliação da expressão: a expressão algébrica a^{2d} deve ser escrita A ↑ (2*D), e não A ↑ 2*D.

Ordem de cálculo das fórmulas (expressões aritméticas)

Quando se tem dúvidas quanto à ordem de avaliação de uma expressão pela máquina, deve-se exprimi-la em instruções separadas, cada uma das quais pode então conter apenas operações simples. Assim, a fórmula do juro simples i = prt/100 pode ser expressa em duas declarações:

170 LET I = P*R*T
180 LET I = I/100

As regras que governam a ordem pela qual as expressões são tratadas pela máquina são dadas em seguida:

1. Quando não existem parêntesis, são realizadas primeiro as exponenciações, em seguida as multiplicações e divisões, e finalmente as somas e subtracções. Assim, na declaração seguinte:

190 LET A = 11 + 2*3 ↑ 4

o valor de A será equivalente a 11 + (2*81), ou seja, 173.

2. Se existe mais do que um operador aritmético ao mesmo nível de tratamento (por exemplo + e —, ou * e /), as operações são realizadas por ordem, da esquerda para a direita.

3. Quando se encontram parêntesis dentro uns dos outros, como acontece no exemplo seguinte, a avaliação realiza-se a partir dos parêntesis interiores, sendo processado em último lugar o conjunto que se encontra encerrado nos parêntesis exteriores. Por exemplo:

200 LET T = 4 + (3*(2 ↑ (3 + 2) — (6*5)))

será processado pela seguinte ordem:

T = 4 + (3*(2 ↑ 5 — 30))
T = 4 + (3*2)
T = 4 + 6 = 10

Os parêntesis podem ser usados para alterar a ordem normal de avaliação dos sinais e operadores aritméticos. Como exemplo desta possibilidade, são mostradas as fases de tratamento de cada uma das expressões que se seguem de modo a permitir ao leitor seguir a ordem de tratamento (experimente estes exemplos no seu equipamento). A tem o valor 2, B é igual a 5 e C é 3.

a) 210 LET Y = A + B*C ↑ 2
Y = 2 + 5*9
Y = 2 + 45
Y = 47

b) 220 LET Y = ((A + B)*C) ↑ 2
Y = (7*3) ↑ 2
Y = 21 ↑ 2
Y = 441

c) 230 LET Y = (A + B)*C ↑ 2
Y = 7*3 ↑ 2
Y = 7*9
Y = 63

d) 240 LET Y = (A + (B*C)) ↑ 2
Y = (2 + 15) ↑ 2
Y = 17 ↑ 2
Y = 289

e) 250 LET Y = (A + B)*(C ↑ 2)
Y = (2 + 5)*9
Y = 7*9
Y = 63

f) 260 LET Y = A + ((B*C) ↑ 2)
Y = 2 + (15 ↑ 2)
Y = 2 + 225
Y = 227

Assim, três operadores podem ser combinados de seis maneiras diferentes. Recorde-se sempre de que se o que acabámos de dizer e as regras de tratamento das expressões aritméticas em Basic lhe parecer demasiado complicado, haverá toda a conveniência em usar as declarações o mais simples possível e realizar apenas uma operação em cada declaração. Os programas são mais fáceis de verificar e corrigir nestas condições. O exemplo da linha 260, por exemplo, pode ser escrito mais simplesmente do seguinte modo:

```
270 LET Y = B*C
280 LET Y = Y↑2
290 LET Y = A + Y
```

Os espaços podem ser usados à vontade entre os operadores, as variáveis e os números no interior de uma expressão aritmética.

Funções

Além dos operadores aritméticos, o Basic permite usar algumas funções para auxiliar a realizar os cálculos de que necessitamos. Muitas destas só têm interesse para os matemáticos e não são indicadas nas versões mais pequenas de Basic que acompanham a maior parte dos microcomputadores em «package». Por exemplo, das funções aritméticas «standard» discutidas neste capítulo, o Tandy Level I Basic apenas fornece a INT, a ABS e a RND, enquanto o Cromenco 3K Basic fornece a ABS e a SGN. O Pet, por outro lado, fornece todas as funções. As variantes das funções fornecidas nas versões de Basic mais usadas são dadas a conhecer no capítulo 7.

As funções mais úteis para o não matemático são:

ABS — Valor absoluto de uma expressão
SGN — Sinal de uma expressão
INT — Maior número inteiro inferior ou igual a uma dada expressão.

A expressão ou *argumento* de uma função é fechada entre parêntesis depois de indicado o nome ou designação da função, por exemplo:

```
300 LET G = ABS (F + 3.7)
```

Estes parêntesis devem encontrar-se presentes em todos os casos. O argumento pode ser uma variável, um número ou qualquer combinação desses ligados por operadores. As expressões entre parêntesis e as próprias funções podem ser utilizados como argumento, pelo que se podem encontrar argumentos com funções, por exemplo:

```
310 LET K = SGN (V)
320 LET W = ABS (T — (D/A))
330 LET B = INT (Q + ABS (X — Z))
```

As funções são avaliadas pela máquina antes de quaisquer operadores existentes nas expressões aritméticas

onde aquelas se encontram. Podem ser combinadas na mesma declaração LET ou PRINT com outros operadores. É melhor evitar introduzir espaços ao digitar o nome da função ou entre este nome e o parêntesis. Podem-se no entanto usar espaços no interior do argumento.

ABS (Y) dá + Y se Y ⩾ 0 e — Y se Y < 0.

SGN(Y) dá + 1 se Y > 0, 0 se Y = 0, — 1 se Y < 0. Nestas condições em «modo calculadora», PRINT SGN (— 17) daria uma saída — 1.

INT (Y) dá o maior número inteiro inferior ou igual ao argumento. Pode dar resultados inesperados no caso de um número negativo a menos que o leitor tenha compreendido que *nunca* pode dar um valor superior ao do argumento; assim, INT (— 2.001) e INT (— 2.999) têm sempre o valor — 3.

Se pretende arredondar um valor para o número inteiro seguinte (de tal modo que 3,5 seria arredondado para 4 e — 3,5 para — 3), pode utilizar a seguinte declaração para um valor guardado em X:

```
340 LET R = INT (X + .5)
```

Usando a declaração anterior pode verificar no seu sistema microcomputador que os seguintes valores de R seriam obtidos a partir dos valores dados a X:

X	R
11.2	11
17.8	18
— 11.2	— 11
— 17.8	— 18
.6	1
.3	0
— .6	— 1
— .4	0

Se deseja arredondar para o decimal seguinte, poderia utilizar a declaração seguinte:

```
350 LET R = INT (10*X + .5)/10
```

Poderá usar o seu equipamento para verificar que se X fosse 5.26, R seria 5.3 e se X fosse 10.78, R seria 10.8.

Se está a trabalhar com uma versão de Basic que não está limitada ao tratamento de números inteiros (como acontece na versão Pet), pode querer obter um dividendo em número inteiro e um resto também em número inteiro. Para não realizar duas vezes a divisão, pode usar a seguinte sequência de declarações para colocar o dividendo inteiro em W e o resto inteiro em R quando X é dividido por Y.

```
360 LET I = INT (X/Y)
370 LET R = X — (Y*I)
```

As funções *matemáticas* são:

SQRT — Dá raiz quadrada. O argumento *deve* ser positivo; em caso contrário obtém-se uma mensagem de erro.

SIN, COS, TAN — O argumento deve encontrar-se em radianos; utilize o factor .0174533 para converter graus em radianos. Para colocar o co-seno de 40 graus em K, a declaração correcta será:

```
380 LET K = COS(40*.0174533)
```

ATN — O arcotangente (o ângulo cuja tangente é) do argumento. Este valor é dado em radianos.

EXP — O antilogaritmo natural ou a exponencial do argumento (se souber o que isto é, saberá como utilizá-lo!)

LOG — O logaritmo natural do argumento. É produzida uma mensagem de erro no caso de o argumento ser zero ou negativo. Se pretende

descobrir o logaritmo na base 10, deve dividir o logaritmo natural pelo argumento do logaritmo natural de 10. Uma declaração para processar $\log_{10}5$ poderia ser:

```
390 LET L = LOG(5)/LOG(10)
```

A função «random» (RND), que se pode encontrar em muitas versões mais completas de Basic, será descrita mais adiante, no capítulo 6.

Entrada simples de dados

Até agora todos os números tem sido comunicados à máquina por declarações de programa, o que tira a flexibilidade aos programas. Se se pretende calcular o quadrado de três números diferentes, é necessário escrever três declarações com a seguinte forma:

```
400 LET X = número*número
```

Isto é obviamente demasiado volumoso para programas reais.

As declarações READ e DATA são usadas em conjunto (nenhuma delas pode surgir num programa sem ser acompanhada pela outra) e constituem uma maneira mais simples de dar entrada a números sem necessidade de utilizar muitas declarações LET. O exemplo seguinte mostra como o seu uso pode dar os valores 11 a 14 às variáveis W a Z de uma maneira mais económica do que utilizando as declarações LET. Usando estas, teremos:

```
410 LET W = 11
420 LET X = 12
430 LET Y = 13
440 LET Z = 14
```


Usando as declarações READ e DATA:

```
450 DATA 11, 12, 13, 14
460 READ W, X, Y, Z
```

A declaração DATA fornece o material necessário para a declaração READ, *em qualquer ponto do programa.*

As declarações DATA podem não se encontrar imediatamente antes da instrução READ, podendo encontrar em qualquer ponto antes da declaração END. O exemplo anterior poderia ser escrito do seguinte modo:

```
470 DATA 11
480 READ W, X
490 DATA 12, 13
500 READ Y, Z
510 DATA 14
```

É mais habitual, no entanto, apresentar as declarações DATA para uma secção do programa antes das declações READ correspondentes. Os elementos de uma lista DATA e as variáveis de uma declaração READ devem ser separados por vírgulas. Os elementos das listas DATA podem não ser expressões.

As listas DATA permitem-nos utilizar o mesmo programa muitas vezes com diferentes conjuntos de dados. Só é então necessário alterar as informações DATA.

A declaração READ utiliza tantos items de dados da lista DATA quantas as variáveis nela incluídas. Pode-se imaginar os elementos de todas as listas DATA colocados numa lista única e um ponteiro que avança de cada vez que um deles é READ («lido») como uma dada variável. É impressa uma mensagem de erro se se lêem (READ) mais elementos do que os fornecidos à máquina na lista DATA.

O uso da declaração READ não destrói nenhum elemento de uma lista DATA. A declaração RESTORE (re-armazenar) coloca o «ponteiro» no início da lista, pelo que todos os elementos podem ser lidos novamente. Assim, no

final da sequência de declarações seguintes (que o leitor pode experimentar no seu equipamento):

```
520 DATA —11, 6, 2, 3
530 READ A, B
540 RESTORE
550 READ C, D
```

verificar-se-á que o valor das variáveis, se se pedir a sua impressão ao computador, será:

```
A = —11
B =   6
C = —11
D =   6
```

Entrada de números durante a execução do programa

Se alterar os valores de cada vez que passa um dado programa, é melhor comunicar estes valores à máquina através do teclado. A declaração INPUT («entrada») permite-lhe comunicar ao computador números que irão formar o conteúdo de uma ou mais variáveis. Depois da execução da declaração

```
560 INPUT A, P, W
```

o seu sistema microcomputador ficará parado até entrar o número apropriado de dados e ser premida a tecla «Return». Muitas vezes, o sistema imprime ou apresenta no écran um sinal interrogativo ou qualquer outro que lhe recordará a necessidade de introduzir dados.

Quando não deu entrada a um número suficiente de dados, o computador apresenta novamente o sinal indicado. A maior parte das versões da Basic ignoram quaisquer dados que você introduza em excesso.

Impressão de texto

Para o guiar a si e a outros utilizadores do seu programa no que se refere a comunicar os dados correctos depois de uma declaração INPUT, é conveniente levar o computador a imprimir uma mensagem apropriada (não se esqueça de que neste livro o termo «imprimir» está também a ser usado para o caso dos sistemas com saída VDU). Qualquer conjunto de símbolos encerrados entre aspas depois da declaração PRINT serão copiados exactamente. Os espaços *no interior* das aspas serão igualmente copiados. A declaração

```
570 PRINT «29 OCTOBER 1978»
```

produzirá na saída:

```
29 OCTOBER 1978
```

A sequência de declarações seguinte mostra uma combinação de declarações PRINT e INPUT:

```
580 PRINT «ESCREVA DOIS NUMEROS»
590 INPUT C, D
```

Esta sequência lembrá-lo-ia, ao utilizar o programa, de que é necessário comunicar à máquina dois números.

Impressão simples

A declaração PRINT acima referida pode também levar o computador a imprimir na saída, isoladas ou em linha, variáveis, números ou o valor calculado de quaisquer expressões. Os elementos de uma lista são impressos num número fixo de «campos» de uma linha se se encontrarem separados por vírgulas. O número exacto de caracteres em cada um dos «campos» ou zonas de impressão varia; algumas versões utilizam cinco zonas de 15 caracte-

res; outras quatro de 15 e uma de 12 caracteres. O número de campos varia também. Numa versão de Basic que possui cinco campos, a saída da seguinte declaração PRINT:

```
600 PRINT 5, 3, 5 + 3,.5 — 3, 5*3
```

seria:

```
5     3     8     2     15
```

A posição exacta dos números no interior dos campos de impressão varia, mas o leitor depressa a aprenderá.

Cada declaração PRINT começa a ser impressa no primeiro campo de impressão de uma linha nova. O uso da expressão PRINT sem incluir mais nada na declaração deixará uma linha em branco na saída.
Pode-se combinar texto com variáveis, separando-o delas por vírgulas. A sequência de declarações seguinte:

```
610 DATA 6, 11, 40
620 READ A, P, W
630 PRINT «O PRODUTO = », A*P*W
```

produzirá:

```
O PRODUTO = 2640
```

As utilizações referidas para a declaração PRINT serão adequadas para os seus primeiros programas. As outras potencialidades desta instrução serão descritas no Capítulo 6.

Um programa completo

Antes de tentar programar para o seu sistema microcomputador os exemplos dados no final deste capítulo.

deverá estudar o exemplo de um programa simples que fornecemos em seguida. Este baseia-se no fluxograma da figura 1.2. Trata-se de um programa para dar entrada no computador de três pesos mensais (não 12 como no programa original) e imprimir a sua média.

```
640 REM PROGRAMA PARA ACHAR A MEDIA
645 REM DE 3 PESOS MENSAIS
650 PRINT «INTRODUZIR 3 PESOS»
660 INPUT X, Y, Z
670 LET A = (X + Y + Z)/3
680 PRINT «A MEDIA =», A
9999 END
RUN
```

A saída poderia ser:

```
INTRODUZIR 3 PESOS
? 70, 74, 69
A MEDIA = 71
```

Experimente agora o exercício seguinte. Sugerem-se algumas soluções no final do livro.

Exercícios 3

1. Dar entrada a dois números para X e Y. Substituir um pelo outro, e imprimir em linhas separadas os seus valores originais e depois de substituídos.
2. Escrever um programa que inverta qualquer número entre 1 e 99 (isto é, 45 transforma-se em 54; 6 em 60). Imprimir os valores originais e invertidos.
3. Usando duas variáveis, escrever um programa que imprima os valores 1017, 43, 1103, 2034, —974.
4. Introduzir um número e imprimir o seu valor, o valor do seu quadrado e o seu cubo. Todas as saídas devem ser feitas sob títulos apropriados.

5. Comunicar ao computador um número representando uma certa quantidade de rodas. Obter na saída o número de conjuntos de quatro rodas incluídos no dado de entrada e a quantidade de rodas que sobram.

6. O reembolso mensal de uma hipoteca de L libras a P % durante N anos é dado pela fórmula:

$$\frac{LR\,(1 + R)^N}{12\,((1 + R)^N - 1)}$$

onde R = P/100.

Escrever um programa que calcula este pagamento mensal.

Os dois problemas que se seguem requerem alguns conhecimentos matemáticos.

7. Faça entrar na máquina o valor de dois lados de um triângulo, X e Y, e o ângulo (em graus) entre eles, C. Calcular e imprimir na saída a área do triângulo, os outros dois ângulos e o terceiro lado.

8. Faça entrar na máquina um número em radianos e obtenha-o na saída sob a forma de um ângulo em graus, minutos e segundos.

4

OPERAÇÕES REPETITIVAS EM BASIC

Até agora o leitor só escreveu programas capazes de exprimir as formas mais simples de fluxograma (em inglês, «flowchart»). Ainda não escreveu quaisquer programas que codificassem fluxogramas com blocos («boxes») em losango, nos quais a sequência do fluxograma é interrompida pela verificação de certas condições, da qual depende a continuação sequencial do tratamento ou o salto para fora da sequência a fim de realizar fases de tratamento específicas, como se descreveu no capítulo 1.

O programa descrito nessas páginas foi resolvido apenas na sua forma mais simples no final do capítulo citado. Por razões de espaço na declaração INPUT, o número de pesos foi reduzido de 12 para três, e o programa exprimia de facto a figura 1.2, não a figura 1.3.

Depois do que lemos no capítulo 1, compreendemos a importância dos *ciclos* (ou *loops*, em inglês), os quais permitem utilizar muitas vezes num mesmo programa as mesmas instruções. Muitos programas consistem de facto em pouco mais do que uma série de ciclos. Para usar esta técnica de programação e programar o fluxograma da figura 1.3 é necessário utilizar a declaração de comparação Basic: IF-THEN (se, então).

A declaração IF-THEN

A forma desta declaração é a seguinte:

IF número / variável / expressão aritmética } relação { número / variável / expressão aritmética

THEN número de linha (da declaração a obedecer se a condição considerada é verdadeira).

O valor de cada lado da comparação é avaliado de tal modo que qualquer das três possibilidades seja comparada com outra. Se a relação é verdadeira, o programa obedece à declaração cujo número de linha é especificado depois do termo THEN. Se não, continua sequencialmente passando à declaração seguinte. O programa divide-se assim em duas partes. As declarações presentes depois da declação THEN são obedecidas por ordem.

Os operadores usados para comparar valores são:

$=$	é igual a
$<>$ ($\neq$)	não é igual a
$>$	é maior do que
$>=$	é maior ou igual a
$<$	é menor do que
$<=$	é menor ou igual a

Note-se que três dos símbolos diferem da notação matemática convencional, consistindo de facto em dois caracteres (por exemplo $>=$), os quais *não* devem em caso algum ser separados por um espaço. Deve-se usar sempre a ordem correcta dos símbolos, ou seja, nunca se deve escrever $=>$ em vez de $>=$.

O problema que consiste em calcular o peso médio, cujo fluxograma foi apresentado no capítulo 1, pode ser agora programado:

```
100 REM PROGRAMA PARA CALCULAR MEDIA
105 REM DE 12 PESOS MENSAIS
110 PRINT «CALCULO DE 12 PESOS MEDIOS»
120 REM K = CONTADOR, T = TOTAL,
125 REM W = PESO INTRODUZIDO
130 LET K = 0
140 LET T = 0
150 REM CICLO PRINCIPAL
160 PRINT «ENTRAR UM PESO EM .KG»
170 INPUT W
180 LET T = T + W
190 LET K = K + 1
200 REM FIM DO CICLO PRINCIPAL
210 IF K < 12 THEN 160
220 REM BOX C1 DO FLUXOGRAMA
230 PRINT «TODOS OS PESOS ENTRADOS»
240 PRINT «MEDIA = », T/12
9999 END
```


Este exemplo utilizou apenas a comparação «$<$». Outras comparações válidas seriam:

```
A = 13
V <> P*Q
N > M + 2
A/3 >= 116
ABS (Z — Y) <=K + G
```

No interior de qualquer expressão, a máquina continua a utilizar a ordem de avaliação já indicada no Capítulo 3.

Tenha cuidado ao usar o símbolo de comparação «=»

As quantidades avaliadas podem diferir muito pouco, devido à representação de quantidades fraccionárias na forma binária que é utilizada internacionalmente em todos os computadores; pode haver uma pequena diferença entre os valores das expressões 10,5 e 10 500/1000, por exemplo.

Se não tem a certeza de que as declarações que estão a ser comparadas serão avaliadas como números inteiros,

convirá usar o método seguinte. Em vez de comparar (X/Y) e Z, quando não se tem a certeza de que ambos serão valores inteiros, deve-se comparar a sua diferença com uma quantidade muito pequena, por exemplo:

ABS (X/Y — Z) < 10E — 20

Esta quantidade deve ser muito pequena relativamente à menor quantidade que pode ser aceite pelo sistema microcomputador considerado. Se a diferença entre os valores que se deseja comparar é tão pequena que em sua opinião pode ser negligenciada e se deve à maneira como o computador representa as fracções (por exemplo, a diferença 10E — 20, que é igual a 0,000 000 000 000 000 000 1), pode-se considerar que as quantidades comparadas são iguais. Se possui uma máquina de calcular electrónica que possa representar os números em «vírgula flutuante» (E), pode experimentar e descobrir uma pequena diferença entre a representação de 10/3 e 60/18.

Convirá recordar o que já foi dito no Capítulo 3 sobre o facto de o cálculo de funções ser um processo relativamente moroso. Nestas condições, se trabalha com uma função (ou com qualquer entidade processada cuja quantidade se mantenha constante de cada vez que o ciclo é realizado) é melhor calculá-la uma vez no exterior do ciclo, armazená-la numa variável e usar este valor no interior do ciclo. Em vez de escrever:

```
250  LET I = 0
260  LET J = 0
270  LET J = J + SQRT (A/B)
280  LET I = I + 1
290  IF I  < 500 THEN 270
9999 END
```

escreva:

```
250  LET L = SQRT (A/B)
260  LET J = 0
```



```
270  LET I = 0
280  LET J = J + L
290  LET I = I + 1
300  IF I < 500 THEN 280
9999 END
```

O valor de SQRT (A/B) é assim determinado apenas uma vez, no exterior do ciclo, em vez de ser calculado 500 vezes no seu interior. Você pode comparar os tempos de execução destes programas no seu próprio sistema, desde que primeiramente introduza os valores A e B.

A declaração STOP

Por vezes, como resultado do teste de uma declaração IF, pode ser necessário terminar a execução do programa apesar de o seu texto continuar no outro ramal da declaração IF. Até agora, os nossos programas «lineares» terminam sempre com a mesma declaração, END (fim), que pode ser usada muitas vezes na maior parte dos sistemas.

A declaração STOP (parar) é por vezes usada nos pontos do programa, além daquele onde se encontra a declaração END, onde se deseja parar a sua execução. Não é normalmente usada nos programas que só têm uma conclusão. A declaração STOP é muitas vezes seguida por uma mensagem que indica a linha onde ocorreu a paragem.

Pode existir qualquer número de declarações STOP num mesmo programa. Os exemplos seguintes ilustram a maneira de a usar. São comunicados à máquina dois números através do teclado, sendo a relação entre eles apresentada na saída.

```
310  INPUT A, B
320  IF A > B THEN 360
330  IF A = B THEN 380
340  PRINT «MENOR DO QUE»
350  STOP
```

```
360 PRINT «MAIOR DO QUE»
370 STOP
380 PRINT «IGUAL A»
9999 END
```

O uso de STOP imediatamente antes de END não é prejudicial, se bem que não tenha qualquer utilidade.

Usos mais importantes da declaração IF

— Contagem de um número de elementos para tratamento idêntico, como acontece no problema de pesos anteriormente referido.

— Conclusão de um processo por identificação de um carácter de entrada que marca o final de uma série de elementos.

— Conclusão de um processo quando a diferença entre um valor em duas execuções consecutivas do ciclo é tão pequena que pode ser considerado negligenciável.

A segunda técnica foi descrita no primeiro capítulo. O carácter que termina uma série de elementos de entrada foi identificado como uma «sentinela». O programa que se segue utiliza a declaração IF para reconhecer esta sentinela; exprime o problema indicado na figura 1.4, onde uma série de elementos, cujo número pode variar de cada vez que o programa é executado pela máquina, é terminada por um zero — um carácter que não poderia ocorrer como entrada válida de pesos.

```
390 REM T = TOTAL, K = CONTADOR, I = ITEM
400 LET T = 0
410 LET K = 0
420 PRINT «INTRODUZA UM NUMERO.»
425 PRINT «INTRODUZA 0 PARA ACABAR»
430 INPUT I
440 LET T = T + 1
450 LET K = K + 1
460 IF I <> 0 THEN 420
```



```
470 REM DECREMENTAR O CONTADOR DE 1
480 PRINT «A MEDIA E», T/(K — 1)
9999 END
```

O uso desta técnica não segue de maneira rigorosa o desenvolvimento do fluxograma da figura 1.4, que se encontra programado literalmente (com o auxílio de uma declaração Basic que está quase a ser descrita) no parágrafo seguinte. Se utilizar o tipo de programação anterior, deve ter presente que na última vez que o ciclo passou foi processada a sentinela (no caso acima, como a sentinela é zero não tem qualquer importância que tenha sido somada ao total) e que a contagem é superior numa unidade ao número de elementos («items»). Vale a pena, quando se comunicam dados à máquina através do teclado, dizer ao operador o que deve ser escrito depois de se ter dado entrada ao último elemento, como se indica na linha 420 do programa.

A terceira técnica interessa particularmente aos matemáticos. As técnicas «iterativas» são vulgares para o cálculo de funções em análise numérica, e a iteração termina quando a diferença entre o valor computado em sucessivas iterações é tão pequena que pode ser considerada negligenciável. O programa seguinte pode ser útil no caso de a versão de Basic por si usada não possuir a função «raiz quadrada». Permite calcular a raiz quadrada (S) de um número apresentado na entrada (*n*) utilizando a fórmula:

$$S = 1/2\,(x + n/x)$$

onde *x* é o S anterior (*x* recebe inicialmente o valor 1). A iteração é continuada pelo ciclo até a diferença entre dois valores sucessivos ser inferior a 10^{-6}.

```
490 PRINT «INTRODUZA UM NUMERO POSITIVO»
500 INPUT N
510 LET S = 1
520 REM INICIO DO CICLO
530 LET X = S
```

```
540  LET S = (X + N/X)/2
550  IF ABS (X — S) > 1E — 6 THEN 530
560  REM FIM DO CICLO
570  PRINT «RAIZ QUADRADA DE», N, «=», S
9999 END
```

A declaração GO TO

Até agora os programas têm sido obedecidos pela máquina segundo a sua ordem de linha, excepto quando a sequência é interrompida devido ao facto de o programa seguir um ramal «THEN» de uma declaração IF. A declaração.

GO TO número de linha

transfere o controlo incondicionalmente para o número de linha indicado depois dos termos GO TO, sendo o programa obedecido sequencialmente a partir desse ponto (a maior parte das versões de Basic aceitam GOTO, ou seja, aceitam a expressão sem qualquer intervalo entre as duas palavras ou com ele). O programa que se segue, que calcula o quadrado e o cubo de um número de entrada e continua até não ser comunicado à máquina mais nenhum número, ilustra o uso desta declaração.

```
580  PRINT «INTRODUZA UM NUMERO»
590  INPUT N
600  PRINT N, N*N, N*N*N
610  GO TO 580
9999 END
```

O controlo pode ser transferido para trás (como acontece no exemplo anterior) ou para a frente; a declaração é válida em ambos os casos.

A declaração GO TO pode ser usada para programar literalmente o fluxograma da figura 1.4 (do qual já se

deu uma versão anteriormente, no exemplo com os números de linha 390-480).

```
620  REM T = TOTAL, K = CONTADOR, I = ITEM
630  LET = 0
640  LET K = 0
650  PRINT «INTRODUZA UM NUMERO»
655  PRINT «INTRODUZA 0 PARA ACABAR»
660  INPUT I
670  IF I = 0 THEN 720
680  LET T = T + 1
690  LET K = K + 1
700  REM O CICLO RECOMECA USANDO GO TO
710  GO TO 650
720  PRINT «A MEDIA =», T/K
9999 END
```

Isto é menos tortuoso do que o programa anteriormente usado para resolver o problema.

Evite GO TO sempre que possível

É legítimo usar a declaração GO TO no exemplo anterior. No entanto, um programa em que o controlo esteja sempre a passar para a frente e para trás, particularmente quando o programa tem muitas páginas, é difícil de compreender e corrigir.

Talvez já tenha encontrado a expressão «Programação estruturada» nas suas leituras. Essencialmente, esta técnica considera que o programa utiliza apenas três tipos básicos de declarações: a *sequência* (uma declaração a seguir à outra), as *condições* (a declaração IF) e os *ciclos* (as declarações IF ou FOR, que dentro em pouco será também definida). Se tivermos cuidado na escolha do operador de comparação e do ponto do ciclo em que se realiza a comparação e se aumenta o contador, pode-se usar THEN em vez de GO TO.

O programa com números de linha 100 a 240, dado mais atrás neste mesmo capítulo, mostra como se pode evitar a declaração GO TO utilizando a comparação K < 12 na linha 210 em vez de K = 12, que à primeira vista pode parecer mais óbvia. Em seguida reescrevemos as últimas linhas do programa utilizando o comparador «=».

```
210  IF K =  12 THEN 230
220  GO TO 160
230  PRINT «TODOS OS PESOS ENTRADOS»
240  PRINT «MEDIA = », T/12
9999 END
```

Os ramais das linhas 210 e 220 cruzar-se-iam no fluxograma (um exemplo daquilo a que alguns chamam «programação tipo spaghetti»...), e esta maneira de concluir parece mais atabalhoada quando comparada com a original. Pense sempre com muito cuidado antes de usar a declaração GO TO.

Por vezes não é fácil determinar a sequência apropriada de declarações em cada ramal de uma comparação IF-THEN. Esta deve ser sempre verificada com dados-amostra quando se faz o fluxograma. A posição da contagem em relação ao teste num ciclo pode alterar a sequência em cada parte do ramal.

Tanto a declaração THEN como a GO TO podem referir-se a números de linha mais atrasados ou adiantados do que o da própria declaração. Convém verificar se não se referem a uma linha que contenha uma declaração «não-executável» como DATA ou REM. Algumas versões de Basic não transferem automaticamente o controlo para a linha que se segue a uma declaração não-executável, e o programa será interrompido então do mesmo modo que nos casos em que o controlo é transferido para uma linha não existente.

Experimente agora os exercícios que se seguem, todos eles necessitando de instruções IF-THEN, no seu próprio

microcomputador. Convirá fazer primeiramente os respectivos fluxogramas.

Exercícios 4

1. Imprimir os números 1 a 100, juntamente com os respectivos quadrados e cubos.
2. Dar entrada a dez números, levando a máquina a calcular e imprimir a sua média.
3. Calcular e imprimir os recíprocos dos números 2 a 100.
4. Usando a taxa de câmbios actual, imprimir uma tabela com o valor das moedas de 10 centavos a 5 escudos e os respectivos valores em libras e marcos.
5. Saem comboios de A para B e C à hora exacta e com intervalos de vinte minutos. Chegam a B 10 minutos e a C 15 minutos depois de partirem de A. Faça um horário entre as 13 e as 16 horas.

Tente resolver os problemas seguintes se aprecia exercícios matemáticos.

6. Apresente à máquina 10 pares de números representando os lados mais pequenos de 10 triângulos rectângulos. Calcule a hipotenusa e imprima na saída o comprimento dos três lados.
7. Calcular 1000 termos da série seguinte, imprimindo o valor de π ao fim de cada 100 termos:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9}$$

8. Calcular e imprimir os termos da série Fibonacci:

$$x_{n+1} = x_n + x_{n-1}$$

entre 1 000 e 1 000 000. Os termos iniciais são 0, 1, 1, 2, 3, 5, 8,...

9. A solução das equações simultâneas:

$$ax + by + c = 0$$

e

$$px + qy + r = 0$$

é dada por:

$$x = (br - cq)/(aq - bq)$$
$$y = (pc - ar)/(aq - bq)$$

Introduza a, b, c, p, q, r, na máquina e imprima na saída os resultados x e y. Faça a máquina imprimir INDETERMINADO no caso de $aq - bq = 0$ e NÃO INDEPENDENTE no caso de $a/p = b/q = c/r$.

5

CICLOS, LISTAS E SUBROTINAS EM BASIC

Todos os ciclos com contadores podem ser programados usando a declaração IF descrita no capítulo 4. A declaração FOR, a que já foi feita uma referência de passagem, constitui no entanto um meio mais económico de tratar os ciclos com contagens, dado que é concebida especificamente para este processo enquanto a declaração IF tem muitas outras utilidades (como seja o teste de erros na introdução de dados) além de controlar os ciclos.

A declaração FOR permite por um lado facilitar a programação (na medida em que necessita de menos instruções, e não é obrigado a preocupar-se com a colocação relativa do incremento do contador e da comparação no interior do ciclo) e aumentar a velocidade a que os programas são executados. Testes realizados em sistemas microcomputadores mostram que o uso da declaração FOR permite velocidades 200 a 2 000 % mais rápidas. Um teste sobre dois ciclos obedecidos 1 000 vezes deu tempos de execução de 12 segundos usando a declaração IF e de 4 segundos usando a FOR.

A declaração FOR

A melhor introdução ao estudo desta declaração consistirá talvez em codificar o fluxograma da figura 1.3 utilizando esta técnica e comparando com o método da declaração IF.

IF

```
100  REM MEDIA DE 12 PESOS
110  REM K = CONTADOR, T = TOTAL
120  REM W = PESO ENTRADO
130  LET T= 0
140  LET K = 0
150  REM CICLO PRINCIPAL
160  PRINT «INTRODUZIR PESO (KG)»
170  INPUT W
180  LET T = T + W
190  LET K = K + 1
200  IF K < 12 THEN 160
210  REM FIM DO CICLO
220  PRINT «TODOS OS PESOS INTRODUZIDOS»
230  PRINT «MEDIA = », T/12
9999 END
```

FOR

```
100  REM MEDIA DE 12 PESOS
110  REM K = CONTADOR, T = TOTAL
120  REM W = PESO ENTRADO
130  LET T = 0
140  REM CICLO PRINCIPAL
150  FOR K = 1 TO 12 STEP 1
160  PRINT «INTRODUZIR PESO (KG)»
170  INPUT W
180  LET T = T + W
190  NEXT K
200  REM FIM DO CICLO
210  PRINT «TODOS OS PESOS INTRODUZIDOS»
220  PRINT «MEDIA = », T/12
9999 END
```

O ciclo começa por uma declaração FOR que define o valor inicial de contagem; de cada vez, o ciclo aumenta este valor de um incremento «STEP», até ser atingido o valor final (depois de TO). A última declaração do ciclo é seguida da declaração NEXT.

A forma completa da declaração FOR é pois:

```
FOR    variável = valor inicial TO valor final STEP
       incremento
NEXT   variável
```

A variável depois de FOR e de NEXT (K no exemplo anterior) é conhecida pelo nome de *variável controlada.* O valor inicial, o valor final e o incremento podem ser números, variáveis ou expressões aritméticas com qualquer grau de complexidade. São válidos os seguintes exemplos:

```
FOR P = Q*3 TO M/L STEP ABS (R +9)
FOR A = T TO 8.05 STEP .05
FOR X2= —J TO —(Y—F) STEP—7
```

A cada declaração FOR deve corresponder uma declação NEXT no final do ciclo. Se o STEP for + 1, pode ser omitido; a linha 150 poderia portanto ser escrita de outro modo:

```
150 FOR K = 1 TO 12
```

Não é porém incorrecto escrever ainda neste caso a parte STEP da declaração.

O terceiro exemplo dado acima mostra um incremento negativo. Quando se usa um incremento negativo deve-se verificar se o valor final é inferior ao inicial, senão a máquina pode executar indefinidamente o ciclo. Isto pode acontecer quando são usadas expressões e não números para valor inicial, valor final ou incremento.

Em seguida observaremos o que acontece quando um ciclo é executado pela última vez se as adições sucessivas do incremento não produzirem exactamente o valor final. Na declaração:

```
160 FOR C = V TO F STEP I
```

o ciclo seria executado pela última vez quando

```
ABS (C) <= ABS (F)
```

No caso de um ciclo positivo:

```
170 FOR K = 2 TO 13 STEP 3
```

o ciclo seria executado tomando K os valores 2, 5, 8, 11. No caso de um ciclo negativo:

```
180 FOR R = 3 TO —9 STEP —4
```

o ciclo seria executado tomando R os valores —3, —7.

Devido à maneira como os microcomputadores representam as quantidades não inteiras, deve-se tomar muito cuidado no caso dos ciclos que utilizam quantidades para a declaração FOR que não sejam inteiras. Na declaração:

```
190 FOR M = 3 TO 3.004 STEP .001
```

há a possibilidade de as adições sucessivas de .001 resultarem num valor levemente superior a 3,004, o que terá como consequência que o último ciclo não será executado. A declaração deve ser escrita assim

```
200 FOR M = 3 TO 3.0041 STEP .001
```

o que assegura que o último ciclo é de facto executado. Se possui um sistema microcomputador que não esteja limitado a trabalho em números inteiros, verifique nele o que acabamos de dizer.

Nunca parta de nenhum princípio em relação ao valor da variável controlada quando abandona um ciclo pela última vez. Se tenciona programar um dado tipo de microcomputador, o manual de programação deste pode dizer-lhe qual é esse valor. No entanto, há algumas variações quanto à maneira como os diversos sistemas realizam esta função, pelo que se tenciona transferir o programa para outro sistema Basic é melhor evitar o uso desta variável assim que abandona o ciclo.

Alguns problemas que podem surgir nos ciclos FOR

Um ciclo FOR pode ser executado:

— Nunca. Algumas versões Basic (mas não a Pet) passam sobre o ciclo se o valor inicial for superior ao final e o incremento for positivo. Isto impede a execução de um ciclo infinito.

— Um número infinito de vezes. Isto pode acontecer quando o valor calculado para o incremento é zero.

— Uma só vez. Isto acontece quando os valores inicial e final são idênticos, o que pode dever-se à maneira como os dados foram introduzidos.

Normalmente, um valor inicial (se for uma variável) pode ser alterado durante a execução de um ciclo, e (se o valor inicial for uma expressão) é possível ainda modificar as variáveis no interior dessa expressão. Em geral: utilize sempre as declarações FOR numa forma simples. Evite modificar qualquer variável no incremento ou no valor final, e pense duas vezes antes de alterar uma variável do valor inicial.

Pode-se usar uma declaração IF ou GO TO para abandonar um ciclo (não voltando a ele), para omitir instruções no interior do ciclo, e para saltar para fora do ciclo e voltar a ele mais tarde. Um exemplo desta última possibilidade poderá ser:

```
210 FOR R = 5 TO 9
220 LET X = R/2
230 IF X <= 3 THEN 370
240 LET T = X/2
250 PRINT T
360 NEXT R
370 LET X = X + 1
380 GO TO 250
```

se bem que fosse menos desajeitado incluir as duas últimas instruções no interior do ciclo. A variável controlada retém o valor que tinha ao saltar para fora do ciclo.

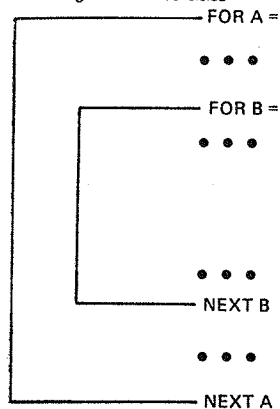
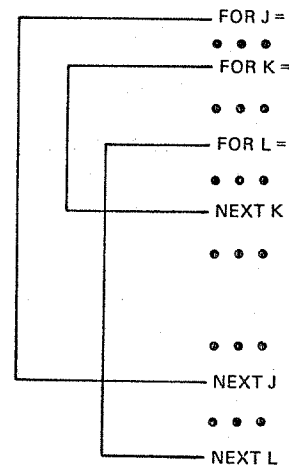
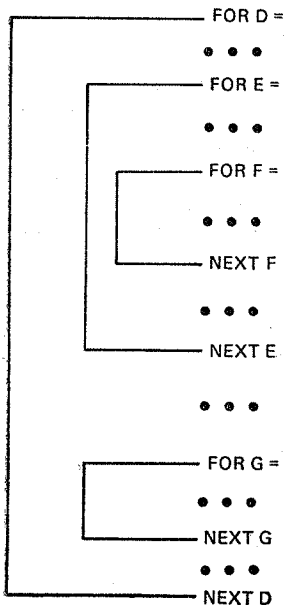
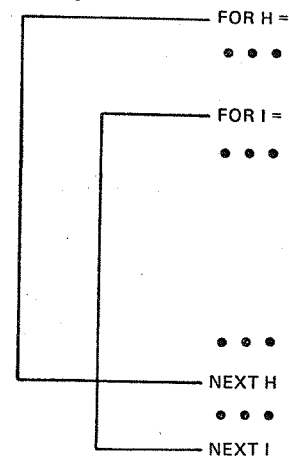
Nunca se deve saltar para meio de um ciclo que não foi abandonado anteriormente, dado que se terá ultrapassado a definição inicial do valor da variável controlada.

Evite alterar o valor da variável inicial, da variável STEP ou da variável controlada durante o ciclo. Em alguns sistemas Basic, não tem grande importância que altere os valores das duas primeiras, dado que são transferidas para contadores antes de o ciclo ser executado pela primeira vez. O valor da variável controlada em caso algum deve porém ser alterada. Quando o ciclo acaba de ser obedecido pela última vez, não é conveniente partir do princípio de que a variável controlada tem um qualquer valor definido. Depressa descobrirá o valor que esta variável tem no seu microcomputador, mas provavelmente verificará que se obtem um valor diferente se se passa o mesmo ciclo com uma configuração diferente.

Estas precauções podem parecer excessivas. Na prática, se você utilizar os ciclos de uma maneira directa não terá razões para se preocupar. Na programação normal nunca sentirá necessidade, por exemplo, de alterar a variável controlada a meio do ciclo ou de saltar para fora deste.

Execução de ciclos no interior de outros («nesting»)

Os ciclos podem ser «aninhados» no interior de outros. O programa seguinte imprime a média de cinco conjuntos de 10 números:

```
390  REM INICIO DO CICLO EXTERIOR
400  FOR J = 1 TO 5
410  LET T = 0
420  REM COLOCA O TOTAL A ZERO
430  REM INICIO DO CICLO INTERIOR
440  FOR N = 1 TO 10
450  PRINT «ESCREVER UM NUMERO»
460  INPUT M
470  LET T = T + M
```

```
480 NEXT N
490 LET A = T/10
500 PRINT «SERIE», J, «A MEDIA = », A
510 NEXT J
9999 END
```

Existe normalmente algum limite (muitas vezes 10) para o número de ciclos que podem ser inseridos no interior de outros, mas isto não é de modo algum limitativo. Nunca deverá usar a mesma variável controlada em mais do que um ciclo interior.

Os ciclos não podem ser «entremeados». A figura 5.1 apresenta exemplos do uso correcto e incorreto dos ciclos interiores.

Velocidade de execução dos ciclos

Verificará que é obrigado a utilizar muitos ciclos nos seus programas, e que nessas condições lhe interessa que aqueles sejam obedecidos pela máquina tão depressa quanto possível. Para obter uma execução rápida, utilize números inteiros sempre que possível para as três variáveis da declaração FOR. A velocidade de execução de um ciclo é também aumentada no caso de você mandar a máquina calcular tantos valores das variáveis quanto possível *uma única vez*, no exterior do ciclo, e não no interior deste de cada vez que a máquina executa a declaração. Por exemplo:

```
520 LET Y = 3.141593/180
530 FOR X = 0 TO 85 STEP 5
540 PRINT «TAN», X, «=», TAN (Y*X)
550 NEXT X
```

é muito mais rápido (dado que Y é calculado apenas uma vez no exterior do ciclo) do que:

```
520 FOR X = 0 TO 85 STEP 5
530 LET Y = 3.141593/180
540 PRINT «TAN», X, «=», TAN (Y*X)
550 NEXT X
```


Exercício 5.1

Realize novamente os problemas 1 a 7 do final do capítulo anterior, mas usando agora declarações FOR. Compare os tempos de execução das duas soluções.

Listas e quadros

Talvez se recorde de que a ideia de um quadro («array») foi introduzida no capítulo 1, como um método de evitar a escrita de dados de entrada uns sobre outros. Um elemento de quadro, ou *variável indexada*, consiste num nome de identificação do quadro seguido de um número ou *índice* que mostra a posição do elemento no quadro ou lista. No capítulo 1 designámo-los por P_1, P_2, P_3, etc., e no fluxograma por P_K (nas figuras usou-se W_K; em inglês, Weight é Peso).

Muitas versões de Basic permitem usar um máximo de 26 quadros numéricos, com os códigos de identificação consistindo numa única letra da gama A a Z. O Tandy Level I Basic, no entanto, permite apenas um quadro. Vejamos algumas variáveis de quadro possíveis:

T(4)   X(K)   V(J + 1)   B(H — F)

O índice pode ser uma expressão aritmética que dê uma resposta inteira, se bem que algumas versões de Basic ignorem qualquer parte fraccionária ou decimal do índice. É melhor usar índices simples: apenas números ou variáveis únicas bastarão na maior parte dos casos.

O índice deve ser sempre escrito entre parêntesis a fim de evitar confusões entre, por exemplo, a variável vulgar D5 e a variável indexada D(5).

Antes de utilizar um quadro, é necessário declarar à máquina a sua dimensão usando uma declaração DIM («dimensão»). A forma desta declaração é:

DIM nome de variável (gama), nome de variável (gama)...

entendendo-se por «gama» o número máximo de elementos da variável quadro.

Nestas condições, a declaração típica DIM apresenta-se:

```
560 DIM X(20), V(45)
```

que reservará a memória necessária para um quadro X com um índice máximo de 20 e um quadro V com um índice máximo de 45. Em algumas, se bem que não em todas, versões de Basic não há necessidade de usar uma declaração DIM para um quadro quando a dimensão deste é 10 ou menos.

Muitas versões de Basic partem do princípio (como acontece neste livro) de que o primeiro elemento do quadro tem o índice 1, pelo que a declaração contida acima na linha 560 do programa permitiria obter espaço para 20 elementos no quadro X. Algumas versões de Basic, no entanto, como o Tandy Basic, consideram que o primeiro elemento de todos os quadros tem o índice 0. A menos que se necessite desesperadamente de espaço no programa é melhor partir do princípio de que todos os quadros se iniciam em 1, mesmo que o microcomputador permita utilizar um índice 0, dado que se torna mais fácil transferir o programa para outro microcomputador.

Deve-se declarar um quadro com uma declaração DIM antes de ser feita qualquer referência a um seu membro num programa. Evitam-se confusões se não se utilizam as mesmas letras para variáveis simples e para quadros.

O programa que se segue é uma expressão da figura 1.5 (capítulo 1), que representa um fluxograma da entrada de 12 pesos e do cálculo da sua média, instruindo ainda a máquina para que imprima o menor peso na

saída. Os pesos são guardados no quadro W, e usa-se K como índice dado que constitui a variável controlada do ciclo. Muitas vezes a variável controlada de uma declaração FOR é usada como índice de um quadro.

```
570 DIM W(12)
580 REM K = CONTADOR, S MENOR PESO
590 REM T = TOTAL
600 LET T = 0
610 LET S = 1000000
620 REM INICIO DO CICLO PRINCIPAL
630 FOR K = 1 TO 12
640 PRINT «INTRODUZIR UM PESO»
650 INPUT W(K)
660 LET T = T + W(K)
670 IF S <= W(K) THEN 690
680 LET S = W(K)
690 NEXT K
700 PRINT «A MEDIA =», T/12
710 PRINT «O MENOR PESO =», S
9999 END
```

A ideia de uma *matriz* como tratando-se de uma espécie de tabela ou quadro com linhas e colunas foi também descrita no capítulo 1. Algumas versões de Basic não permitem usar quadros com mais de uma dimensão; outros não colocam qualquer limite ao número de dimensões do quadro.

Talvez o leitor recorde que um elemento de matriz foi definido como tendo *dois* índices, o primeiro indicando a linha e o outro a coluna. Em Basic, as matrizes podem ser declaradas na mesma declaração DIM do quadros vulgares ou em declarações separadas. Uma típica declaração de matriz será:

```
720 DIM A(2,6), B(4,5)
```

que declara a matriz A com duas linhas e seis colunas, e a matriz B com quatro linhas e cinco colunas. Não se pode

usar a mesma letra para dois quadros de dimensões diferentes. As observações anteriores quanto ao uso do índice 0 em algumas versões de Basic aplicam-se igualmente a quadros com mais do que uma dimensão.

O programa seguinte lê uma matriz com seis linhas e oito colunas e imprime a soma de cada linha:

```
730  DATA (24 números)
740  DATA (24 números)
750  DIM E(6,8)
760  REM T = TOTAL DA LINHA
762  REM M = CONTADOR LINHAS
765  REM N = CONTADOR COLUNAS
770  FOR M = 1 TO 6
780  LET T = 0
790  FOR N = 1 TO 8
800  READ E(M,N)
810  LET T = T + E(M,N)
820  NEXT N
830  PRINT «LINHA», M, «O TOTAL = », T
840  NEXT M
9999 END
```

Experimente agora os exemplos seguintes, todos eles necessitando de variáveis de quadro.

Exercício 5.2

1. Ler 10 números de uma lista de dados. Imprimir a sua média e o número com o maior desvio absoluto relativamente à média.

2. Divida cada elemento de uma tabela com cinco linhas e quatro colunas pelo maior elemento desta.

3. Imprima o maior elemento de uma tabela com três linhas e quatro colunas, e a linha e a coluna desse elemento.

4. Imprima os números primos entre 3 e 100.

5. Imprima um quadrado de cinco linhas e cinco colunas, contendo cada linha e coluna apenas uma vez cada um dos números um a cinco.

6. Ler de uma lista de dados não mais de 60 números positivos, terminando com — 1. Imprima o maior, o menor e a média.

Funções e subrotinas

Já verificou certamente que as funções standard lhe permitem não escrever secções complicadas do programa de cada vez que as deseja utilizar; seria laborioso e consumiria muito espaço escrever linhas de programa sempre que desejasse usar TAN e ABS, por exemplo.

Muitas versões de Basic permitem-lhe usar uma declaração DEF para escrever linhas de programa definindo funções que não fazem parte de um sistema dado, ou tratamentos que necessita de usar frequentemente. Está limitado a 26 destas funções «definidas pelo utilizador», e deve ser capaz de exprimir todas as funções numa linha única. Esta última limitação impede-o de fazer declarações de funções complexas (excepto no caso de algumas versões de Basic que permitem o emprego de funções multi-lineares).

O nome da função por si definida consiste na designação FN seguida de uma letra: FNA, FNG, FNT, por exemplo. Na declaração DEF o nome da função é seguido de uma variável muda, depois «=» seguida de uma expressão aritmética que deve ser computada pela função. Uma função típica (adicionar 15 por cento de IVA a um preço) poderia ser:

```
850 DEF FNV(P) = P + .15*P
```

Cada função deve ser escrita antes de ser usada. A escrita de uma declaração DEF não permite automaticamente realizar a operação. A função deve ser chamada

ou invocada no programa quando o valor que nos interessa substitui a variável muda. O programa curto seguinte lê dois preços do teclado e imprime o seu valor mais 15 por cento de imposto sobre o valor acrescentado.

```
860  DEF FNV(P) = P + .15*P
870  PRINT «INTRODUZIR PRIMEIRO PRECO»
880  INPUT X
890  PRINT «PRECO + IVA = », FNV(X)
900  PRINT «INTRODUZIR SEGUNDO PRECO»
910  INPUT Y
920  PRINT «PRECO + IVA», FNC(Y)
9999 END
```

Compreende-se agora o interesse da variável muda, que é substituída por X e Y nas linhas 890 e 920 respectivamente. Qualquer expressão aritmética pode ser colocada entre parêntesis (ou ser o *argumento*) quando se invoca uma dada função. Por exemplo:

```
FNV(P + D),    FNV(P — 20),    FNV(P*Q)
```

constituirão formas válidas de chamar a função «imposto sobre o valor acrescentado» de que estivemos a falar.

O sistema *não* deveria alterar o valor de uma variável representada pela mesma letra da variável muda. Infelizmente, algumas versões recentes do Basic fazem-no. Verifique o que acontece no seu caso, acrescentando ao programa do valor acrescentado as seguintes linhas:

```
855 LET P = 29
930 PRINT P
```

Se passar o programa em seguida pode verificar se P retém o seu valor de 29. Se tal não acontecer sabe que nos seus programas nunca pode usar uma variável com a mesma letra da variável muda.

Pode-se usar uma função definida pelo utilizador em todos os locais onde usaria uma expressão aritmética, por exemplo:

```
IF G < FNV(E + Y) THEN 950
FOR J = T TO FNH(E) STEP 6
LET Q = X + ABS(FND(M) + I)
```

Pode-se fazer referência a variáveis além da muda durante a escrita de uma declaração de função. Na sequência de declarações seguintes:

```
940 DEF FNQ(J) = A*J + B*J + C
950 DATA 2, 3, 4, 5
960 READ A, B, C, D
970 PRINT FNQ(D)
```

o sistema deveria imprimir: (10 + 15 + 4) = 29. Quando usa variáveis diferentes da muda numa declaração de função DEF, no entanto, torna mais difícil a transferência da função para outros programas.

Deve evitar que a função se invoque a si própria (o elegante funcionamento que os especialistas de computadores por vezes designam de «recursividade») dado que a maior parte das versões não o permitem.

Subrotinas

As funções definidas pelo utilizador podem ser consideradas como pequenas subrotinas, capazes de serem expressas numa linha única, e limitadas a processos aritméticos. As subrotinas foram descritas, no capítulo 1, como a possibilidade de escrever uma só vez uma sequência de declarações muito usada, usá-la tantas vezes quantas se quiser no programa, e ter sempre a certeza de voltar à declaração do programa que se segue ao chamamento de subrotina.

Em Basic, uma subrotina é uma sequência de declarações que termina pela declaração RETURN, e que pode surgir em qualquer ponto do programa. Uma subrotina é invocada pela declaração:

GOSUB número de linha onde a subrotina se inicia

Para assegurar que se inicia a subrotina através da declaração GOSUB, convém escrever todas as subrotinas depois do **STOP do programa** principal e imediatamente antes de END. Para auxiliar a transferência das suas subrotinas de um programa para outro, utilize números de linhas elevados (neste livro iniciamos as subrotinas em 8000) e nomes de variáveis e quadros não normalmente utilizados.

Em seguida observamos um programa feito para o fluxograma da figura 1.8 (capítulo 1). São lidos dois números do teclado. A sua média e as médias dos seus quadrados e cubos são computadas e impressas através de uma subrotina.

```
980  PRINT «INTRODUZIR DOIS NUMEROS»
990  INPUT X, Y
1000 LET A = X
1010 LET B = Y
1020 GOSUB 8000
1030 LET A = X*X
1040 LET B = Y*Y
1050 GOSUB 8000
1060 LET A = A*X
1070 LET B = B*Y
1080 GOSUB 8000
1090 STOP
1100 INICIO DA SUBROTINA
8000 LET M = (A + B)/2
8010 PRINT «MEDIA = », M
8020 RETURN
9999 END
```


Não se deve fazer qualquer referência numa subrotina (através de uma declaração IF-THEN ou GOTO) a uma linha no seu exterior, dado que se impede deste modo a sua utilização noutros programas. Qualquer declaração DIM ou DEF no interior de uma subrotina aplicar-se-á ao longo de todo o programa. Uma declaração RETURN pode ser usada muitas vezes no interior de uma subrotina, nos pontos onde se pretende abandoná-la e voltar ao programa principal.

Uma subrotina pode invocar outra, se bem que a maior parte das versões de Basic não permitam a uma subrotina chamar-se a si mesma. Uma subrotina pode chamar outra escrita antes ou depois no texto do programa. Existe normalmente um limite para esta inclusão de subrotinas no interior de outras, mas nunca é tal que interfira na programação normal.

A subrotina que se segue lê uma série de números terminando em 9999 e imprime o número de elementos, o seu total e a média. Utiliza outra subrotina de uso geral para assegurar que não é realizada a divisão por zero (nos casos em que a subrotina está a tratar elementos negativos e positivos mistos, por exemplo temperaturas). Imprime uma mensagem afirmando que o número verificado é zero e coloca um indicador (Z) em zero.

A subrotina principal é introduzida em 8500. As declarações REM devem ser sempre usadas nas subrotinas para indicar as letras variáveis por elas usadas; quando se escreve o programa principal pode-se evitar estas, ou verificar se possuem conteúdos apropriados para a subrotina. Antes de começar a programar convém decidir qual das subrotinas disponíveis desejamos usar, de modo a não haver depois confusão quanto às letras variáveis.

```
8000 LET Z = 1
8010 REM T = NUMERO A SER TESTADO
8015 REM Z = INDICADOR
8020 REM Z = ZERO SE NUMERO IGUAL A ZERO
8030 IF T <> 0 THEN 8060
8040 LET Z = 0
```

```
8050 PRINT «O NUMERO TESTADO IGUAL A ZERO»
8060 RETURN
8500 LET U = 0
8510 REM ARRAY = Y, TOTAL = T,
8512 REM U = NUMERO DE ITEMS
8515 REM ITEM EM USO = Q
8520 LET T = 0
8530 DIM Y(100)
8540 PRINT «INTRODUZIR UM ELEMENTO»
8545 PRINT «ULTIMO ELEMENTO = 9999»
8550 INPUT Q
8560 IF Q = 9999 THEN 8610
8570 LET U = U + 1
8580 LET Y(U) = Q
8590 LET T = T + Y(U)
8600 GO TO 8540
8610 GOSUB 8000
8620 PRINT «NUM. DE ELEMENTOS =», U,
8625 PRINT «TOTAL =», T
8630 IF Z = 0 THEN 8650
8640 PRINT «A MEDIA = », T/U
8650 RETURN
```

Exercício 5.3

1. Escreva subrotinas para:

(*a*) converter uma quantidade em milímetros para metros, centímetros e milímetros;
(*b*) calcular a média de uma série de 10 números guardados num quadro;
(*c*) aplicar um desconto de 2,5 % se um preço é superior a 5 000$00 e 5 % se é superior a 10 000$00;
(*d*) converter escudos em libras esterlinas, à taxa de câmbios actual.

2. Escrever uma subrotina que imprima o maior e o menor números de um quadro.

3. Escrever uma subrotina que imprima os factores positivos de qualquer número inferior a 100, e use-a com o elemento de entrada de um programa completo.

Se se interessa por matemáticas, experimente os exercícios seguintes.

4. Escrever uma subrotina que resolva a equação do 2.º grau $ax^2 + bx + c = 0$, usando o método da fórmula, e imprimir o resultado.
5. Escreva uma subrotina que converta um número em radianos num ângulo em graus, minutos e segundos.
6. Escreva as funções que realizam as operações seguintes, utilizando cada uma delas num programa em três elementos de uma lista de dados. Imprima os resultados.

(*a*) logaritmo decimal;
(*b*) tangente a partir do argumento em graus;
(*c*) área de um círculo a partir do raio;
(*d*) conversão de um ângulo em graus para radianos;
(*e*) arredondamento de um número positivo, por exemplo, 1,5 transforma-se em 2;
(*f*) raiz cúbica.

6

ALGUMAS POSSIBILIDADES ADICIONAIS DO BASIC

As características discutidas até agora são comuns a quase todas as versões de Basic incluindo o «Basic reduzido», que utiliza uma memória muito pequena (no Capítulo 7 registam-se algumas das omissões de certas variantes). Este capítulo descreve algumas características menos comuns mas que se podem encontrar em muitos microcomputadores e que serão familiares aos leitores que tenham usado Basic em minicomputadores ou em computadores grandes. Dado que estas características são menos habituais, existe uma variação correspondentemente maior na maneira como são utilizadas.

Tratamento de caracteres

Até agora só trabalhámos com números. O único contacto do leitor com dados não numéricos foi no caso da declaração «PRINT», que lhe permite imprimir mensagens e títulos. Não conseguiu ainda introduzir na máquina caracteres através do teclado ou de listas de dados, e não soube comparar ou escolher caracteres. Estas potencialidades, com uma importância cada vez maior dado o tratamento bastante extenso de textos e de palavras nos negócios e o aumento da quantidade de informações em letras (incluindo receitas e listas de endereços) podem

ser muito úteis ao amador, que certamente desejará também guardar palavras na memória do seu computador e levá-lo a tratá-las.

Os caracteres são considerados em Basic como sendo mantidos numa *cadeia* (string), que é uma sequência de caracteres tratada como uma unidade. Já encontrámos estas *strings* na declaração PRINT, pois os caracteres no interior das aspas constituem de facto uma cadeia. Um elemento de uma cadeia pode ser qualquer carácter (incluindo um espaço) do conjunto de caracteres existentes na configuração que estamos a utilizar no computador.

As cadeias de caracteres são guardadas em *variáveis de cadeia,* que são indicadas por uma letra seguida de um sinal $. A dimensão da cadeia que pode ser guardada numa única variável varia consideravelmente (entre 15 e 4096 em algumas versões vulgares de Basic). O manual de programação que acompanha o seu computador dar-lhe-á as indicações necessárias quanto a isto. Se deseja poder usar os seus programas em sistemas diferentes do seu, convém usar apenas cadeias com um máximo de 15 caracteres. Algumas versões pequenas de Basic não permitem o uso de variáveis de cadeia, ou então só permitem usar uma. Algumas versões de Basic utilizam quadros: A$(1) indicará o primeiro membro de um quadro de cadeia A$ (não existe qualquer relação entre uma variável ou um quadro *numericos* e uma variável *de cadeia,* ou um quadro de cadeia com a mesma letra, pelo que seja qual for a utilização que se dê a W nada será alterado no conteúdo da variável de cadeia W$).

Algumas versões de Basic permitem uma «cadeia nula», que não contém quaisquer caracteres e pode ser estabelecida pelo comando LET:

```
100 LET R$ = « »
```

Isto é normalmente usado para limpar o espaço usado por uma cadeia comprida, nas versões de Basic que permitem o armazenamento de longas cadeias de caracteres numa variável única.

Os valores dos caracteres (que podem ser um único carácter) podem ser introduzidos em variáveis de caracteres por uma declaração LET (como acima), uma declaração DATA (na qual se podem misturar dados numéricos e caracteres, desde que estejam separados por aspas) ou uma declaração INPUT:

```
110 LET T$ = «PARADISE»
120 DATA 5, «LOST», 7, «BY JOHN MILTON»
130 READ U, V$, X, Y$
140 INPUT Z$
```

Nos exemplos anteriores, T$ conterá PARADISE e V$ LOST, Y$ By JOHN MILTON e Z$ aquilo que for introduzido através do teclado. Tal como na declaração PRINT, os espaços são importantes no interior da cadeia: são contados no número de caracteres e são guardados (como acontece na variável Y$ citada).

As variáveis de cadeia não podem ser usadas para guardar números para cálculos numéricos, pelo que a declaração:

```
LET G = J$ + B$
```

seria incorrecta (tal como as variáveis numéricas não podem ser usadas para guardar caracteres). Uma cadeia pode evidentemente conter números ou símbolos não usados aritmeticamente, por exemplo:

```
160 LET F$ = «JANUARY 1980»
170 LET H$ = «* — * — *»
```

Pode-se obter a saída de caracteres de uma cadeia recorrendo à declaração PRINT, tal como acontece na seguinte sequência de declarações

```
180 LET Q$ = «LORNA»
190 LET J$ = «OXLEY»
200 PRINT Q$, J$
```

que apresentará na saída:

```
LORNA          OXLEY
```

Tal como na declaração READ, a declaração PRINT pode guardar caracteres numéricos e alfabéticos misturados; por exemplo, a sequência de declarações:

```
210 DATA 11, 6, «ANNE»
220 READ A, B, C$
230 PRINT A, B, «1940», C$, «PATRICIA»
```

daria na saída:

```
11      6      1940      ANNE      PATRICIA
```

Não é possível produzir qualquer trabalho útil usando informação em caracteres a menos que o computador possa escolher e identificar uma cadeia (como acontece no caso em que se escreve «SIM» (YES) ou «NAO» (NO) em resposta a uma pergunta apresentada pelo computador). Para tornar possíveis estas comparações, dá-se aos caracteres alfabéticos uma ordem de valor, ou *sequência de verificação*, que variará em função do sistema microcomputador que se está a usar. Em qualquer caso $A < Z$ e $0 < 9$, e o espaço é normalmente considerado como o carácter mais pequeno; mas existem diferenças no valor dos símbolos e dos sinais de pontuação. A sequência de declarações seguintes mostra uma comparação simples de cadeias:

```
240 PRINT «QUER CONTINUAR»
250 PRINT «ESCREVER SIM OU NAO»
260 INPUT L$
270 IF L$ = «NAO» THEN 9999
(resto do programa)
```

Se forem comparadas duas cadeias de comprimento diferente, será usada toda a cadeia mais curta e a parte

correspondente da mais comprida. Se se verificar que a cadeia e a parte de cadeia em causa são iguais, a cadeia mais curta é considerada a mais pequena. Assim, na sequência de declarações seguinte, é impresso 2 na saída:

```
280 DATA «ANNE», «ANN»
290 READ V$, G$
300 IF V$ > G$ THEN 330
310 PRINT «1»
320 STOP
330 PRINT «2»
```

Será necessário verificar o valor de um espaço na sequência de verificação se se está a usar uma técnica de escolha letra-a-letra em vez de palavra-a-palavra, o que permite saber se «ABBEY WOOD» é considerado maior ou menor do que «ABBEYFIELD».

Algumas versões de Basic possuem funções de cadeia como LEN(Z$), que dará o número de caracteres contidos na variável Z$. Estas funções não serão discutidas aqui, dado que variam bastante nos diferentes sistemas.

Outras possibilidades de saída

As várias opções da declaração PRINT diferem bastante nos diversos sistemas Basic. As características discutidas neste parágrafo são no entanto comuns à maior parte deles.

Se deseja deixar uma linha em branco no papel ou no écran video, pode fazê-lo utilizando a palavra PRINT isolada. A sequência de declarações seguinte:

```
340 DATA 11, 5, 62
350 READ A, B, C
360 PRINT «O ANIVERSARIO DE JULIA E A»
370 PRINT
380 PRINT A, B, C
```

dará na saída:

```
O ANIVERSARIO DE JULIA E A

11        5        62
```

Até agora a saída foi restringida pelas zonas de impressão (tal como no exemplo anterior, em que os três números se encontravam bastante separados). A maior parte das versões de Basic possuem diferentes comprimentos de campo: uma versão possui cinco colunas de 15 posições cada. No entanto, as zonas ou campos de impressão podem ser alteradas se se usar um ponto e vírgula em vez de uma vírgula. O número de espaços entre cada elemento quando se usa a separação por ponto e vírgula altera-se. Uma versão introduz dois espaços se os elementos são numéricos, um espaço se um deles é numérico e o outro uma cadeia de caracteres, e nenhum espaço quando ambos os elementos são cadeias.

Experimente no seu sistema; depressa verificará qual o espacejamento exacto produzido quando se usa o ponto e vírgula ou a vírgula para separar elementos impressos. Se a declaração PRINT contém em qualquer dos modos mais elementos do que os que podem ser impressos numa linha única, a impressão continuará a ser automaticamente realizada na linha seguinte.

Nas declarações seguintes podemos observar a diferença entre estes dois tipos de separador:

```
390 DATA 6, 60, 600, 6000, 60000
400 READ V, W, X, Y, Z
410 PRINT V, W, X, Y, Z
420 PRINT V; W; X; Y; Z
430 PRINT «ISTO», «E», «UM», «EXEMPLO»
440 PRINT «ISTO»; «E»; «UM»; «EXEMPLO»
```


dará na saída:

```
6            60            600            6000            60000
6  60  600  6000  60000
ISTO           E             UM             EXEMPLO
ISTOEUMEXEMPLO
```

Se deseja imprimir na saída números contendo fracções decimais, é mais conveniente usar o separador de vírgula, no qual os pontos são automaticamente alinhados uns sob os outros.

Se terminar a declaração PRINT com uma vírgula ou um ponto e vírgula, a impressão será automaticamente (se houver espaço suficiente) continuada na mesma linha, como se mostra no exemplo seguinte:

```
450 DATA 1, 2, 3, 4
460 READ A, B, C, D
470 PRINT A, B,
480 PRINT C, D
490 PRINT A; B;
500 PRINT C; D
```

dará a saída:

```
1            2            3            4
1  2  3  4
```

Os dois tipos de separadores podem ser misturados na mesma declaração PRINT. Na sequência anterior, se a declaração seguinte fosse:

```
505 PRINT «OS RESULTADOS SAO»; A, B, C
```

a saída seria:

```
OS RESULTADOS SÃO 1         2         3
```

Na maior parte das versões de Basic encontra-se a função TAB. A sua forma é:

TAB (expressão em números inteiros)

A expressão em números representa a posição do carácter, contando a partir da margem esquerda. Se a sequência de declarações anteriores continuasse:

505 PRINT TAB(8); A; TAB(20); B

a saída seria:

1 2

A função TAB é anulada pelos formatos «vírgula» e «ponto e vírgula», pelo que é necessário ter a certeza de que o argumento é superior ao número de espaços que a sua versão de Basic destina normalmente para estes formatos. Pode-se usar a função TAB para alinhar os elementos se se está a usar o modo ponto e vírgula, e para imprimir gráficos grosseiros numa tele-impressora se não se possui um VDU ou símbolos gráficos.

Ficheiros

A maior parte das versões de Basic permitem a constituição de ficheiros; possibilitam ler e escrever em discos ou fitas magnéticas, tornando assim possível usar informação previamente acumulada. As instruções para utilização dos discos magnéticos variam muito, pelo que este parágrafo concentra-se nos ficheiros em cassettes de fita magnética. Talvez já as possa ter utilizado para guardar programas, caso em que se utiliza o comando SAVE.

Antes de se usar um ficheiro de dados ou informações em Basic o ficheiro deve ser «aberto», mesmo que vá

escrever um ficheiro novo numa fita virgem. Uma forma da declaração OPEN (abrir) é:

OPEN número de ficheiro lógico, número de dispositivo físico, opção de entrada/saída (input/ /output), nome do ficheiro.

Fornece-se um *número de ficheiro lógico* no programa, e utiliza-se este número quando se pretende escrever ou ler esse ficheiro. Existe normalmente um limite quanto ao número de ficheiros lógicos que podem ser usados num único programa.

O *número de dispositivo físico* (por exemplo número do leitor de cassette se está a usar um sistema que utiliza mais do que um) será especificado no manual do sistema em uso.

A *opção de entrada/saída* define se pretendemos ler um ficheiro (e impedir qualquer futuro utilizador do programa de escrever nesse ficheiro), escrever nele, ou ler e escrever simultaneamente. Estas opções podem controlar o posicionamento de marcas de fim de fita e de fim do ficheiro.

Existe normalmente um limite quanto ao número de caracteres alfabéticos que se podem usar no *nome do ficheiro*. Convém verificar se as letras de abertura de dois ficheiros diferentes não são idênticas, dado que a procura do nome mais curto pode nem sempre conduzir à escolha do ficheiro correcto.

No final de um programa é necessário fechar cada ficheiro. A declaração apropriada para este efeito é:

CLOSE número de ficheiro lógico

Conforme a opção usada na declaração OPEN correspondente, são escritas marcas de fim de ficheiro ou de fim de fita. Não é possível realizar qualquer tratamento num ficheiro depois de ter sido fechado, pelo que é necessário ter cuidado a fim de assegurar que não se introduz uma declaração CLOSE num ciclo que se pre-

tenda usar várias vezes. Se não fechamos o ficheiro, pode ser mutilado.

Os ficheiros são lidos pela declaração INPUT, cuja forma para leitura do ficheiro é:

INPUT # número de ficheiro lógico, variáveis

(o sinal # é substituído por dois pontos em algumas versões de Basic). Um programa curto para ler os primeiros três elementos de um ficheiro e imprimi-los na saída poderia ter a forma seguinte:

```
510 OPEN 1, 1, 0, JANSTATS,
520 INPUT # 1, A, B, C
530 PRINT  «O INICIO DO FICHEIRO E»; A, B, C
540 CLOSE 1
```

Se a sua versão de Basic lhe permite usar a declaração INPUT (e não GET) para obter informações de um ficheiro, deve verificar se inclui a variável conveniente depois da palavra INPUT.

Os ficheiros são escritos usando uma forma da declaração PRINT; em muitas versões esta palavra é seguida por #, número de ficheiro lógico. Existem muitas vezes certas limitações quanto ao número de elementos (em Pet, 80) que se podem escrever num ficheiro em qualquer momento sem um retorno do carro. A sequência de declarações seguinte mostra um título e 80 elementos escritos num ficheiro:

```
550 OPEN 1, 1, 2, DECSTATS
560 PRINT # 1, «VENDAS DEZEMBRO 1979»
570 PRINT «INTRODUZIR 80 ELEMENTOS»
580 FOR J = 1 TO 80
590 INPUT A
600 PRINT # 1, A
610 NEXT J
620 CLOSE 1
```


Como é óbvio, este parágrafo sobre tratamento de ficheiros é bastante genérico dado que os pormenores da utilização de ficheiros variam bastante com as diversas variantes de Basic. Os programas que contêm instruções de ficheiros raramente podem ser completamente transferidos de um microcomputador para qualquer outro.

A função RND

A maior parte das versões de Basic incluem a função RND para a produção de números aleatórios («random»). Estes são muito úteis se você escreve programas para simular jogos de azar, ou para escrever um modelo de alguma situação em que se observem acontecimentos casuais. Os números verdadeiramente aleatórios tornariam quase impossível repetir um dado ciclo de números, o que tornaria impossível a repetição idêntica de qualquer experiência e produziria dificuldades no que se refere ao teste e correcção de programas.

A forma clássica da função número aleatório é:

RND(X)

onde X é um argumento inteiro. Assim, uma declaração do tipo:

```
630 LET R = RND(1)
```

produziria uma fracção positiva aleatória entre 0 e 1.

Num dado sistema, a sequência de declarações que se segue:

```
640 FOR K = 1 TO 3
650 PRINT RND(1)
660 NEXT K
```

produzirá a saída:

```
.2435041     .2998482     .6075527
```

Os números produzidos são de um modo geral característicos de cada sistema.

Infelizmente, em muitas versões de Basic há uma grande variedade de argumentos acrescentados (se o forem) a RND, assim como nos números produzidos. No Pet, um argumento negativo, produz uma nova série de números aleatórios; um argumento 0 dá o último número aleatório produzido, enquanto um argumento maior do que zero produzirá um número aleatório novo entre 0 e 1.

Alguns sistemas não necessitam de qualquer argumento, e o NIBL (National Industrial Basic Language) utiliza dois argumentos devolvendo um número inteiro compreendido entre eles. Neste livro será sempre utilizado o argumento 1, partindo-se do princípio de que é neste caso produzida uma fracção decimal positiva. O manual do sistema que estiver a usar fornecerá os detalhes necessários quanto à maneira como as variações do argumento produzirão diferentes resultados e como é possível repetir uma dada sequência de números pseudo-aleatórios.

Um método de alterar a sequência consiste em introduzir no teclado uma variável de cada vez que o programa é passado, a fim de indicar quantos números pseudo-aleatórios devem ser ignorados antes de serem usados:

```
670 PRINT «ESCREVER QUANTOS NUMEROS»
675 PRINT «ALEATORIOS A IGNORAR»
680 INPUT K
690 FOR J = 1 TO K
700 LET Y = RND(1)
710 NEXT J
```

Se a sequência anteriormente referida (640-660) fosse usada com as declarações acima e se introduzisse «2» através do teclado, o primeiro número aleatório que se obteria depois de abandonar o ciclo seria .6075527.

Se a sua função RND produz uma fracção, esta pode ser convertida em inteiro multiplicando por uma potência adequada de 10. A sequência de declarações:

```
720 FOR K = 1 TO 50
730 PRINT 100*RND(1)
740 NEXT K
```

produziria 50 números aleatórios na gama 0 — 100.

Se deseja simular o lançamento de uma moeda ao ar, a declaração seguinte dará 0 ou 1 (caras ou coroas) com uma probabilidade de 0,5:

```
750 LET C = INT(RND(1) + .5)
```

Se deseja imitar o lançamento de um dado com seis resultados igualmente prováveis, a declaração deverá ser:

```
760 LET D = INT(6*RND(1)) + 1
```

Outras possibilidades

A maior parte das possibilidades discutidas neste capítulo apresentam-se de maneiras bastante diferentes em várias versões Basic. Por outro lado, algumas versões desta linguagem possuem muitas outras potencialidades, como por exemplo:

— Funções de matriciação;
— Funções de cadeias de caracteres;
— Declarações PEEK e POKE para trabalho em código-máquina;
— Possibilidades gráficas;
— Instruções para controlo do cursor.

Estas características não são tão vulgares como as discutidas neste capítulo.

Experimente agora os exercícios indicados em seguida.

Exercícios 6

1. Ler cinco palavras tiradas de uma lista de dados, ordená-las e imprimi-las por ordem alfabética.
2. Introduzir um número, e imprimir na saída o resto obtido quando se divide o número em causa por 11.
3. Ler 10 números inteiros menores do que 30, e imprimi-los na saída colocando ao lado de cada um deles uma fiada de sinais + indicando o valor de cada um dos números; por exemplo:

5 +++++

4. Simule 100 lançamentos de um dado, e imprima na saída o número de vezes que cada uma das faces fica virada para cima.
5. Simule a tiragem de 13 cartas ao acaso de um mesmo baralho, e imprima na saída a «mão» obtida.
6. Uma roleta tem impressos os números 0 a 36 (ignorar o 0). Simule 1000 rotações da roleta, e imprima na saída o número de vezes que se obtêm pares, ímpares, Marque (1-18) e Passe (19-36).

7

VARIANTES DE BASIC

A linguagem Basic nunca foi definida com o mesmo rigor da Cobol ou da Fortran. Nos capítulos 3 a 6 referiu-se a existência de certas características como a capacidade para processar certas cadeias de caracteres e o número de quadros permitidos. Muitos fornecedores de compiladores ou interpretadores Basic possuem ainda outras características desde que usem um espaço razoável de memória para o «software» de tradução, existindo nestas condições numerosos «Basic ampliados», a maior parte dos quais possui características únicas. Por outro lado, as versões de Basic que utilizam quantidades de memória muito reduzidas (os Basic «pequenos») omitem certas características «standard», como a capacidade para manipular números com parte decimal. Algumas variações do Basic devem-se ao número de bytes usado para guardar números e à maneira como as fracções decimais são representadas.

As tabelas deste capítulo contêm informações sobre algumas versões de Basic que podem ser encontradas em sistemas microcomputadores. Estas tabelas devem funcionar como guias quando está a escolher o sistema microcomputador a adquirir, sendo extremamente importante que o escolhido possa realizar certas funções como o tratamento de números com uma parte decimal. As versões descritas já estão em uso há algum tempo, não

sendo portanto provável que as suas características se alterem radicalmente.

Os capítulos 3 a 6 contêm informações sobre o que pode ser definido como o «núcleo comum» das implementações de Basic em microcomputadores. Foram feitos muitos acrescentos a este núcleo comum, como por exemplo:

— A capacidade para omitir LET na declaração de atribuição;
— Múltiplas declarações numa única linha;
— Funções com caracteres alfabéticos;
— Uso de AND e OR numa declaração IF;
— Comandos para saídas gráficas.

A referência da tabela a «todas as funções» significa que a versão em causa contém as seguintes funções:

RND, ABS, SGN, INT, SIN, COS, TAN, ATN, LOG, EXP, SQR

A referência a «todos os operadores aritméticos» indica a inclusão de

↑ + — * /

A referência a «todos os operadores de relação» indica a inclusão de:

= <> >= > <= <

	Altair/MITS: Disk Extended	*Apple Integer*	*Applesoft*
1. Só inteiros	Não	Sim	Não
2. Variáveis de cadeia	Sim	Sim	Sim
3. Funções de cadeia	Sim	Sim	Sim
4. Todas as funções	Sim	Só RND, SGN, ABS	Sim
5. Todos os operadores aritméticos	Sim	Sim	Sim
6. Todos os operadores de relação	Sim	Sim	Sim
7. Instruções para a elaboração de matrizes	Não	Não	Não
8. Funções definidas pelo utilizador	Sim	Sim	Sim
9. Quadros multi-dimensionais	Sim	Não	Sim
10. Facilidades gráficas	Não	Sim	Sim
11. PEEK e POKE	Sim	Sim	Sim
13. Número máximo	32 767	32 767	
13. Tratamento de ficheiros	Sim		Sim
14. Possibilidades ou restrições adicionais	Formatos de saída versáteis Diagnósticos Ficheiros de dados em disco	Gráficos a cores sem READ DATA Diagnósticos Cadeia de programas	Gráficos a cores incompatível com INTEGER Operadores Lógicos

	Crofton Electronics: NIBL	*Crofton Electronics: 8K disc interactive Basic*	*Cromemco: Control Basic*
1. Só inteiros	Sim	Não	Sim
2. Variáveis de cadeia	Sim	Sim	Sim
3. Funções de cadeia	Não	Sim	
4. Todas as funções	RND, MOD, únicas aritméticas	Sim	SGN, ABS, RND
5. Todos os operadores aritméticos	Não ↑	Sim	Não ↑
6. Todos os operadores de relação	Sim	Sim	Sim
7. Instruções para a elaboração de matrizes	Não	Não	Não
8. Funções definidas pelo utilizador	Não	Não	Não
9. Quadros multi-dimensionais	Sem quadros	Só dois	Nenhum
10. Facilidades gráficas	Não	Não	Não
11. PEEK e POKE	Operador indirecto tem esta função	Sim	Não
13. Número máximo	32 767	9,999E+99	32 767
13. Tratamento de ficheiros	Não	Sim	Não
14. Possibilidades ou restrições adicionais	Um «Basic pequeno» permite declarações múltiplas na mesma linha	Ficheiros de dados iterativos Possibilidade de montagem Auxiliares de diagnóstico	Ocupa 3K Pode chamar subrotinas em código - máquina

	Cromemco: 16K Basic	*Computer Software Services: Super Basic*	*CW 6800: 3K Basic*
1. Só inteiros	Não	Não	Sim
2. Variáveis de cadeia	Sim	Sim	Não
3. Funções de cadeia	Sim	Sim	Não
4. Todas as funções	Sim	Sim	ABS, INT, RND, SGN
5. Todos os operadores aritméticos	Sim	Sim	Não ↑
6. Todos os operadores de relação	Sim	Sim	Sim
7. Instruções para a elaboração de matrizes	Não	Não	Não
8. Funções definidas pelo utilizador	Sim	Sim	Não
9. Quadros multi-dimensionais	3 dimensões	2 dimensões	Não
10. Facilidades gráficas	Não	Sim	Não
11. PEEK e POKE	Sim	Sim	Não
13. Número máximo	9,99E+62	9,99E+99	32 767
13. Tratamento de ficheiros	Sim	Sim	Não
14. Possibilidades ou restrições adicionais	Podem-se abrir 8 ficheiros em qualquer momento Programas podem ser encadeados Diagnóstico	Os comandos podem ser abreviadas Os programas podem ser encadeados	Não READ, DATA

	CW 6800 8 K Basic	*Midas M100 Basic*	*MVT Basic (Altair)*
1. Só inteiros	Não	Não	Não
2. Variáveis de cadeia	Sim	Sim	Sim
3. Funções de cadeia	Sim	Sim	Sim
4. Todas as funções	Sem ATN	Sim	Sem EXP
5. Todos os operadores aritméticos	Sim	Sim	Sim
6. Todos os operadores de relação	Sim	Sim	Sim
7. Instruções para a elaboração de matrizes	Não	Não	Não
8. Funções definidas pelo utilizador	Sim	Sim	Sim
9. Quadros multi-dimensionais	Sim	Sim	Sim
10. Facilidades gráficas	Não	Sim	Não
11. PEEK e POKE	Sim	Sim	Não
13. Número máximo		9E+18	
13. Tratamento de ficheiros	Não	Sim	Sim
14. Possibilidades ou restrições adicionais	Tratamento de periféricos	Editor de caracteres Declarações múltiplas numa só linha Diagnósticos Tratamento de ficheiros em disco	Compilador Diagnósticos Ficheiros de acesso não-sequencial

	North Star Extended Basic 5-PFB	*Pet*	*Research Machines: Tiny Basic interpreter*
1. Só inteiros	Não	Não	Sim
2. Variáveis de cadeia	Sim	Sim	Não
3. Funções de cadeia	Sim	Sim	Não
4. Todas as funções	Sim	Sim	ABS, RND
5. Todos os operadores aritméticos	Sim	Sim	Dois
6. Todos os operadores de relação	Sim	Sim	Sim
7. Instruções para a elaboração de matrizes	Não	Não	Não
8. Funções definidas pelo utilizador	Sim	Sim	Não
9. Quadros multi-dimensionais	Sim	Sim	Não
10. Facilidades gráficas	Não	Sim	Sim
11. PEEK e POKE	Sim	Sim	Não
13. Número máximo	Rigor de 14 dígitos	1,701411838E+38	32767
13. Tratamento de ficheiros	(Inversão 6)	Sim	Não
14. Possibilidades ou restrições adicionais	Definição de múltiplas funções do utilizador na mesma linha Operadores lógicos Possibilidade de renumerar o programa	Teclado de símbolos gráficos Declarações múltiplas na mesma linha Diagnósticos Possibilidade de edição no visor	Declarações múltiplas na mesma linha Ordens abreviadas As expressões podem ser substituídas por valores

	Research machines: 9K Basic	*Research machines: 12K Basic*	*SOL: Basic 5*
1. Só inteiros	Não	Não	Não
2. Variáveis de cadeia	Sim	Sim	
3. Funções de cadeia	Sim	Sim	Não
4. Todas as funções	Sim	Sim	Sem ATN
5. Todos os operadores aritméticos	Sim	Sim	Sim
6. Todos os operadores de relação	Sim	Sim	Sim
7. Instruções para a elaboração de matrizes	Não	Não	Não
8. Funções definidas pelo utilizador	Sim	Sim	Não
9. Quadros multi-dimensionais	Sim	Sim	Não
10. Facilidades gráficas	Sim	Sim	Não
11. PEEK e POKE	Sim	1E + 38	Não
13. Número máximo	1E+38	Sim	0,9999E+127
13. Tratamento de ficheiros	Sim	Sim	Sim
14. Possibilidades ou restrições adicionais	Edição «on-LINE» Diagnóstico Duas saídas independentes	O mesmo que o anterior, mais: Produção automática de números de linha Funções multi-lineares	Edição Inclusão de 6 subrotinas umas nas outras

	SOL: Extended Cassette Basic	*Sorcerer Standard Basic*	*Tandy TRS80 Level I Basic*
1. Só inteiros	Não	Não	Sim
2. Variáveis de cadeia	Sim	Sim	DOIS
3. Funções de cadeia	Sim	Sim	Não
4. Todas as funções	Sim	Sim	ABS, RND
5. Todos os operadores aritméticos	Sim	Sim	Não ↑
6. Todos os operadores de relação	Sim	Sim	Só =><
7. Instruções para a elaboração de matrizes	Sim	Não	Não
8. Funções definidas pelo utilizador	Sim	Sim	Não
9. Quadros multi-dimensionais	Sim	Sim	Não
10. Facilidades gráficas	Sim	Sim	Sim
11. PEEK e POKE	Sim	Sim	Não
13. Número máximo	0,9999E+127		1E + 38
13. Tratamento de ficheiros	Sim	Não	Não
14. Possibilidades ou restrições adicionais	Diagnósticos Partes do compilador podem ser tiradas da memória Renumeração automática Operadores lógicos	Símbolos gráficos definidos pelo utilizador Caracteres alfanuméricos de ordem superior e inferior	Só 1 quadro Palavras de comando abreviadas

	Tandy TRS80 Level II Basic
1. Só inteiros	Não
2. Variáveis de cadeia	Sim
3. Funções de cadeia	Sim
4. Todas as funções	Sim
5. Todos os operadores aritméticos	Sim
6. Todos os operadores de relação	Sim
7. Instruções para a elaboração de matrizes	Não
8. Funções definidas pelo utilizador	Sim
9. Quadros multi-dimensionais	Sim
10. Facilidades gráficas	Sim
11. PEEK e POKE	Sim
13. Número máximo	1,701411E + 38
13. Tratamento de ficheiros	Sim
14. Possibilidades ou restrições adicionais	Incompatível com nível I Disponibilidade de variáveis de dupla precisão 900 nomes de variáveis 5 Tipos de variável

8

INTRODUÇÃO À PROGRAMAÇÃO EM LINGUAGEM ASSEMBLER E EM CÓDIGO-MÁQUINA

Este capítulo é uma introdução às características úteis da programação em linguagem assembler e em código-máquina, e não propriamente um guia detalhado da programação de um microcomputador específico. Descreve alguns aspectos que serão úteis, mas não trata em profundidade qualquer das mais intrincadas potencialidades de certas pastilhas microprocessadoras.

Se está a usar um sistema microcomputador em «package» para os seus negócios, trabalho doméstico ou cálculo científico, é improvável que prefira este tipo de linguagem à Basic. Se, no entanto, deseja ligar o seu sistema a um dispositivo de controlo externo ou a algum periférico não estandardizado, como um conversor ou amplificador analógico-digital, será obrigado a escrever a interface apropriada em linguagem assembler ou em código-máquina. A existência de declarações PEEK e POKE na maior parte das versões de Basic em sistemas em «package» parece indicar uma certa consciência da necessidade de usar o código-máquina em certas ocasiões.

Não há dúvidas de que os programas em código-máquina são bastante mais rápidos do que os em linguagem de alto nível, mas esta velocidade de execução deve ser comparada com o tempo ocupado com a escrita e o desenvolvimento do programa, devendo-se ainda ter em conta o facto de um programa Basic com acrescentos em código-máquina não ser transferível para sistemas

baseados no uso de uma outra pastilha microprocessadora. No entanto, não existe nada intrinsecamente difícil na programação numa linguagem de baixo nível. Até ao início dos anos 60, quando a linguagem Fortran ficou completamente desenvolvida, a maior parte dos programas técnicos eram escritos em linguagem assembler. Esta linguagem era também bastante usada para a programação de aplicações comerciais até ao início da mesma década, tendo sido treinados nela milhares de programadores. Não se deixe portanto desiludir — a programação em linguagem assembler e em código-máquina não é de modo algum tão difícil como pode parecer.

Talvez seja útil ler novamente as páginas 33 a 40 do capítulo 2, a fim de recordar parte da terminologia usada. Isto permitir-lhe-á recordar também o que são os sistemas binário e hexadecimal. Os exemplos deste capítulo serão escritos em linguagem assembler, a partir da qual é possível obter o código máquina consultando o manual de funcionamento do sistema que utiliza e, nos locais apropriados, indicando um endereço de sua escolha.

Por exemplo, quando se usa o código Intel 8080, se se está a trabalhar com um programa de pagamentos e se armazenar o total computado no endereço 260, a tradução terá a seguinte aparência:

```
STA PAY 00110010 (STA)
        00000100 }
        00000001 } endereço 260 (ordem invertida)
```

As razões que conduzem à inversão da ordem do endereço são discutidas na parte apropriada deste capítulo. Se, como é óbvio, se tem a sorte de dispor de um assemblador para o nosso sistema, pode-se encarregá-lo de realizar esta tradução.

Como já se mencionou no capítulo 2, se necessitamos de trabalhar em código-máquina, é muito útil escrever e verificar o programa primeiramente em linguagem assembler, antes de converter as instruções para binário ou hexadecimal para entrada por teclas ou interruptores.

Os exemplos deste capítulo limitar-se-ão às quatro pastilhas microprocessadoras mais vulgares, tanto no seu próprio equipamento como nos sistemas em «package». As pastilhas em causa (seguidas de alguns dos sistemas em «package» que as utilizam) são: Intel 8080 (Altair e Imsai); Motorola 6800 (SWTP e MSI); MCS6502 (Apple e Pet); e Zilog Z-80 (Tandy TRS80 e Research Machines). Estes microprocessadores têm muito em comum; todos apresentam 40 pernes de ligação e trabalham com operandos de oito bits, havendo muitos aspectos comuns nos respectivos manuais de utilização.

Alguns aspectos fundamentais

1. Notação

As notações binárias e hexadecimais foram descritas mais atrás. Por vezes as listas de endereços ou programas são fornecidas em *octal*. Este sistema utiliza a base 8. Os números decimais 1 a 10 correspondem em octal a:

1 2 3 4 5 6 7 10 11 12

2. Byte

Todos os microcomputadores discutidos utilizam uma unidade de oito bits, conhecida por *byte*, como unidade de armazenamento e como comprimento básico da instrução.

3. Endereço

Um *endereço* é o número de identificação de um byte de memória (uma «posição de memória») que possui dados ou uma instrução. O exemplo da secção anterior referia-se a dados mantidos no endereço 260.

Se bem que um endereço se refira a um byte único, os endereços dos sistemas microcomputadores têm nor-

malmente um comprimento de dois bytes (16 bits), pelo que é possível referenciar posições com endereços superiores a 255. Teoricamente o maior endereço que pode ser guardado em dois bytes é 65 535, mas o seu sistema tem provavelmente uma quantidade de memória muito mais reduzida.

Afirma-se muitas vezes que a memória de um dado sistema possui um certo número de «K», onde K indica 1024 bytes; nestas condições, um sistema de 32K terá de facto 32768 posições, com endereços que variam entre 0 e 32767. Note-se que as primeiras 256 posições da memória são muitas vezes descritas como a *página 0* da memória.

Se você está a usar linguagem assembler, existe uma instrução ORG através da qual é possível definir o endereço inicial do programa, isto é,

```
ORG 2000
```

iniciará a assemblagem no endereço 2000.

Você deve assegurar que os endereços por si escolhidos para o armazenamento dos programas e dos dados não interferem com os usados por qualquer programa monitor ou subrotina que se encontre em memória ao mesmo tempo que o seu programa.

4. Registos

Estes são usados para guardar dados e para alguns objectivos especiais. Dado que constituem uma parte integrante do microprocessador, o acesso a eles é muito mais rápido do que o acesso às posições de memória. Devem portanto ser usados, quando não o estão a ser para qualquer função especial, para o armazenamento de resultados intermédios.

Todos os sistemas microprocessadores aqui tratados possuem os registos seguintes:

— Contador de programa (indica o endereço da instrução seguinte);

— Apontador da pilha de dados (objectivo indicado no capítulo 9);
— Acumulador ou registo A (8 bits).

Exemplos de outros registos:

	8080	6800	6502	Z-80
Função geral				
	B,C	B,C	Nenhum	B,C
	D,E,H,L			D,E,H,L, A', B', C', D', E', H', L',

8080	6800	6502	Z-80
Funções especiais			
Nenhuma	Registo índice X	Reg. ind. X, Y	IX, IY Valor de interrupção Renovação da memória

O uso dos registos de funções especiais será explicado neste mesmo capítulo.

5. Flags

Uma «flag» é um bit que define uma condição especial (por exemplo o «overflow» aritmético no acumulador) como sendo verdadeira ou falsa. Recebe o valor 1 se a condição se encontra presente e 0 se não se encontra. As «flags» são usadas principalmente no tratamento aritmético e nas interrupções, e são discutidas nas secções apropriadas deste capítulo e do seguinte.

Transferência de dados

Algumas das instruções mais importantes respeitam à deslocação de dados de registos para posições de memó-

ria e vice-versa. O tratamento aritmético é realizado no acumulador, pelo que é necessário uma transferência para este registo. Os resultados temporários são muitas vezes movidos do registo para a memória, enquanto a entrada e a saída de dados obriga normalmente a uma transferência de memória para um registo.

Transferências entre a memória e o acumulador

Para se realizarem as operações aritméticas, é necessário que um dos operandos seja carregado no acumulador usando uma instrução LDA (ou equivalente), e é ainda necessário que o resultado seja armazenado usando uma instrução do tipo STA. Em seguida fornecem-se as instruções apropriadas para os quatro microprocessadores. Na sua utilização, como aliás acontece no caso das instruções fornecidas noutras secções deste capítulo, é necessário saber o comprimento da instrução, as «flags» afectadas (se houver alguma) e o formato em código-máquina, consultando o manual.

8080

LDA Carrega o acumulador com o *conteúdo* do endereço indicado nos dois bytes que se seguem ao código da operação. A parte do endereço de menor ordem é indicada antes da de maior ordem; isto não afecta a designação que damos ao endereço em linguagem assembler, mas se se está a trabalhar em código-máquina é importante indicar bem esta ordem (ver o exemplo de uso da instrução STA no primeiro parágrafo deste capítulo).

LDAX B Carrega no acumulador o *conteúdo* do endereço dado pelos registos B e C. Isto permite-nos usar a mesma instrução para indicar um endereço diferente, alterando simplesmente o conteúdo dos registos.

LDAX D Como acima, mas usando os registos D e E.

STA Armazena o conteúdo do acumulador no endereço fornecido pelos dois bytes que se seguem ao código da operação. Corresponde a LDA.

STAX B e STAX D correspondem a LDAX B e LDAX D.

6800

CLR Leva uma posição de memória ao valor 0.

CLRA Leva o acumulador a 0.

LDAA Carrega no acumulador o *conteúdo* do endereço dado nos dois bytes que se seguem ao código de operação. A parte de maior ordem do endereço surge antes da parte de menor ordem (ao contrário do que acontece no 8080); se o endereço se encontra na gama 0-255 só é necessário um byte para o armazenar (isto aplica-se a todas as instruções que contêm um endereço de memória). Pode também carregar um *valor* no acumulador: o valor é normalmente precedido pelo sinal #, por exemplo LDAA # 5, e é indicado no segundo byte da instrução. Esta forma de endereçamento é conhecida pelo nome de *endereçamento imediato*.

STAA A ordem de «armazenar» correspondente a LDAA.

6502

LDA e STA são semelhantes à LDAA e STAA do 6800, excepto no facto de a representação de um endereço em dois bytes ser feita de «trás para a frente», como no 8080.

Z-80

LD A, (endereço da posição ou nome)	Funciona como a LDA do 8080
LD (endereço da posição ou nome), A	Funciona com a STA do 8080
LD A, (BC)	Funciona como a LDAX B do 8080

LD A, (DE)	Funciona como a LDAX D do 8080
LD (BC), A	Ordem de «armazenar», inversa da LD A, (BC)
LD (DE), A	Ordem de «armazenar», inversa de LD A, (DE).

Tenha cuidado com a vírgula e os parêntesis em todas estas instruções.

Podemos observar em seguida um programa que utiliza parte das instruções referidas para transferir entre si os conteúdos de duas posições de memória, designadas COX e BOX. As instruções «load» não afectam o conteúdo da posição de memória que transferem para o acumulador, e as instruções «store» não afectam o conteúdo da posição transferida.

8080	*6800*	*6502*	*Z-80*
LDA COX	LDAA COX	LDA COX	LD A, (COX)
STA DUMP	STAA DUMP	STA DUMP	LD (DUMP), A
LDA BOX	LDAA BOX	LDA BOX	LD A, (BOX)
STA COX	STAA COX	STA COX	LD (COX), A
LDA DUMP	LDAA DUMP	LDA DUMP	LD A, (DUMP)
STA BOX	STAA BOX	STA BOX	LD (BOX), A

Transferência de dados envolvendo outros registos

8080

LHLD	Carrega nos registos L e H *respectivamente* os conteúdos de duas posições de memória: o endereço dado nos dois bytes que se seguem ao código de operação, e o mesmo endereço mais um. Esta instrução é muito útil para a transferência de um endereço para L e H.
SHLD	Armazena o conteúdo dos registos L e H num par de posições de memória consecutivas.
XCHG	Troca os conteúdos dos pares de registos D, E e H, L.
MOV R1, R2	Desloca o conteúdo do registo R2 para R1. Por exemplo, MOV A, E transferiria o conteúdo do registo E para o acumulador (considerado registo A).
MOV M, R	Desloca o conteúdo de um registo para a posição de memória *definida pelo endereço armazenado nos registos L e H*. Por exemplo, MOV M, A, transferiria o conteúdo do acumulador para o endereço dado nos registos L e H.
MOV R, M	Inverte o processo anterior, ou seja, desloca o conteúdo de uma posição de memória para um registo. Por exemplo, MOV A, M transferiria para o acumulador o conteúdo do endereço dado nos registos L e H.
LXI RP	Carrega um par de registos (BC, DE, HL) com o valor de dois bytes que se seguem ao código de operação. Por exemplo, LXI B, COX carregaria os registos B e C com o endereço de COX.
MVI M	Desloca o valor do byte consecutivo ao código de operação para a posição de memória especificada pelos registos L e H. Por exemplo, MVI M, colocaria 7 no endereço apropriado.
MVI R	Como acima, mas desloca o valor para um registo em vez de o fazer para uma posição de memória. Por exemplo, MVI A,6 colocaria um 6 no acumulador.

6800

LDAB	Carrega no registo B o conteúdo do endereço dado nos dois bytes que se seguem ao código de operação.
STAB	Instrução de armazenamento correspondente.

TAB Transfere o conteúdo do acumulador para o registo B.
TBA Inverso da operação anterior.
LDX Carrega no registo de índice o conteúdo de duas posições de memória: o endereço dado nos dois bytes que se seguem ao código de operação, e esse endereço mais um.
STX A correspondente instrução de armazenamento.
TPA Coloca todas as «flags» no acumulador.

6502

LDX Carrega o registo de índice X a partir de posições de memória (como a LDX do 6800).
STX Instrução de armazenamento correspondente.
TAX Coloca o conteúdo do acumulador no registo X.
TXA Coloca o conteúdo de X no acumulador.
LDY, STY, TAY, TYA são as instruções correspondentes para o registo de índice Y.

Z-80

Apresentam-se em seguida as instruções de transferência mais comuns.

LD R1, R2 Carrega o conteúdo do registo R2 em R1. R1 e R2 podem ser quaisquer dos registos A a E, H e L.
LD R, n *n* é o *valor* que se encontra no byte que se segue ao código da operação. Por exemplo, LD A, 0 limpará o acumulador.
LD R, (HL) Carrega o conteúdo do endereço definido nos registos H e L para um registo especificado. Por exemplo, LD C, (HL) colocaria o conteúdo do endereço definido nos registos H e L no registo C.
LD (HL), R Instrução inversa da anterior.
LD (HL), n Carrega o conteúdo do byte que se segue ao código da operação no endereço definido pelos registos H e L.

LD A, (BC)
LD (BC), A
LD A, (DE)
LD (DE), A
} Semelhantes a LD R, (HL) e LD (HL), R. Carregam ou armazenam o acumulador do ou para o endereço especificado pelos registos DE ou BC.
LD HL, nn Carrega os dois bytes que se encontram depois do código de operação nos registos H e L. As instruções LD BC, nn e LD DE, nn, são semelhantes a esta.

Adição e subtracção

Só são fornecidas instruções para a adição e a subtracção. A multiplicação e a divisão devem ser realizadas por subrotinas, que de um modo geral são fáceis de obter.

O tipo mais simples de aritmética é o executado em binário, envolvendo dois valores inteiros de um único byte. É esta a base das operações aritméticas realizadas com números maiores.

A notação binária já foi descrita no Capítulo 2. A representação de números negativos, no entanto, não foi discutida aí. Na maior parte das aplicações é provável que se encontrem quantidades negativas (como um débito ou uma temperatura baixa), e se você possui um sistema que apresenta o conteúdo de registos e da memória em luzes colocadas sobre interruptores pode ver-se perante um número negativo. Em binário, os números negativos são representados na notação «complemento para dois». Esta utiliza o dígito mais significativo (o da extrema esquerda) de um número binário como o «dígito de sinal», que indica se o número é positivo ou negativo. O dígito em causa tem o valor 0 no caso de um número positivo e o valor 1 no caso de um negativo. Isto limita o maior número positivo que se pode incluir num único byte a 127 (01111111), e o maior número negativo a — 128 (10000000).

Para determinar a representação negativa de um número positivo podem-se empregar dois métodos:

1. Trocar todos os 0s por 1s e os 1s por 0s, adicionando em seguida 1. Por exemplo:

+ 7 = 00000111
— 7 = 11111001

2. Subtrair de 2 levantado à potência do número de bits da representação que se está a usar. Se se usa um byte usaremos 2^8 ou 256; se se usam dois bytes, 2^{16} ou 65 536. Por exemplo:

256 — 7 = 249
donde — 7 = 11111001

Pode-se verificar esta conversão somando o número positivo e a sua conversão negativa; deve dar zero em todos os bits que se está a usar para representação do número, por exemplo:

	+ 7	00000111
mais	— 7	11111001
		(1)00000000

Os dois métodos indicados darão ainda o equivalente positivo de qualquer número negativo que se observe nas luzes em binário, por exemplo:

	11110011
Inverter e somar um:	00001101 = 13

Portanto, o número era — 13. Vejamos outras representações negativas num único byte:

— 1	11111111
— 3	11111101
— 4	11111100
— 64	11000000

Os resultados de uma adição ou subtracção encontram-se normalmente no acumulador. Todos os sistemas considerados podem adicionar o valor do byte que se segue ao código de operação, pelo que no caso de uma instrução (no 8080):

ADI 20

obter-se-ia a soma de 20 ao conteúdo do acumulador; 20 é conhecido pelo nome de *operando imediato.*

Os quatro sistemas considerados possuem uma flag de transporte (as «flags» foram já brevemente discutidas). Esta flag assume um valor 1 se ocorre o transporte de uma unidade («ou um») e passa a zero se tal não acontece. Todos os sistemas discutidos excepto o 6502 possuem instruções separadas para adição e subtracção tendo ou não em conta a flag. Esta função é útil numa aritmética de «multi-precisão», discutida mais adiante. As instruções de adição e subtracção são as seguintes:

8080

ADI, ACI	Adiciona o conteúdo do byte que se segue ao código de operação ao conteúdo do acumulador — com ou sem transporte respectivamente.
SUI, SBI	Subtracção realizada nas mesmas condições — com ou sem transporte.
ADD R, ADC R	Adiciona o conteúdo de um registo ao acumulador — com ou sem transporte.
SUB R, SBB R	Subtracção nas mesmas condições — — com ou sem transporte.
ADD M, ADC M	Adiciona o conteúdo de uma posição de memória indicada pelos registos L e H — com ou sem transporte.
SUB M, SBB M	Subtracção realizada nas mesmas condições — com ou sem transporte.

6800

ADDA, ADCA	Adiciona um operando imediato ou o conteúdo de uma posição de memória ao acumulador, com ou sem transporte.
SUBA, SBCA	Subtracção. realizada nas mesmas condições.
ADDB, ADCB / SUBB, SBCB	Realiza as mesmas funções usando o registo B em vez do acumulador.
ABA, SBA	Adiciona/subtrai o conteúdo do registo B ao/do acumulador, mantendo o resultado neste.

6502

ADC, SBC Adiciona/subtrai um operando imediato ou o conteúdo de uma posição de memória ao/do acumulador, com transporte. CLC limpa a flag de transporte, se se deseja assegurar que o resultado não seja influenciado por este.

Z-80

ADD n, ADC n, SUB n, SBC n (onde *n* é um operando imediato) assemelham-se às instruções ADI, ACI, SUI, SBI no sistema 8080.

ADD r, ADC r, SUB r, SBC r correspondem à instrução tipo ADD R do sistema 8080.

ADD (HL), ADC (HL), SUB (HL), SBC (HL) correspondem à instrução tipo ADD M do sistema 8080.

O exemplo seguinte determina a diferença entre as duas variáveis CAT e DOG, adicionando em seguida 10 e 20. Considera-se que todos os números e a soma resultante podem ser guardados num único byte. Não se usa a possibilidade de realizar transporte. Para evitar isto no sistema 6502 é necessário passar ao valor 1 o transporte numa subtracção e passá-lo a zero antes de executar uma adição. Nos outros sistemas utilizam-se algumas instruções

diferentes para ilustrar a maneira de as usar e portanto este pequeno programa não constitui necessariamente a maneira mais eficaz de realizar o cálculo. Os programas 8080 e Z-80 devem deslocar uma soma para um registo e o endereço de uma posição de memória para os registos H e L antes de realizar o cálculo.

8080	*6800*	*6502*	*Z-80*
MVI B, 10	LDA B # 10	LDA CAT	LD B, 10
LXI H, DOG	LDAA CAT	SEC	LD HL, DOG
LDA CAT	SUBA DOG	SBC DOG	LD A, (CAT)
SUB M	ABA	CLC	SUB (HL)
ADD B	ADDA # 20	ADC # 10	ADD B
ADI 20		ADC # 20	ADD 20

Se se está a trabalhar em linguagem assembler, em vez de escrever um programa nesta linguagem e em seguida convertê-lo para código, pode-se utilizar as potencialidades do sistema discutidas em seguida. EQU permite dar um valor a uma variável antes de esta ser usada numa instrução do programa. Esta possibilidade é muito útil para definir constantes frequentemente usadas, por exemplo

DUZIA EQU 12

DB tem uma função semelhante (e é por vezes escrita como DEFB), por exemplo

DUZIA DB 12

DS reserva espaço para armazenamento de um número especificado de bytes indicados por um nome, por exemplo

QUANT DS 4

reservará 4 bytes para a variável QUANT.

Aritmética de multi-precisão

Você não desejará ver-se limitado a quantidades não superiores a 127. Os sistemas 8080 e Z-80 possuem instruções para aritmética de dois bytes nos registos. Se deseja trabalhar com quantidades superiores àquela, é necessário utilizar a possibilidade «transporte» em instruções como a ADC.

A instrução de adição de dois bytes em 8080 é DAD seguida de B, D ou H, e adiciona aos registos H e L o conteúdo dos pares de registos BC, DE e HL. O primeiro registo indicado em cada caso conterá o bit de sinal (0 se for positivo, 1 se for negativo) e a parte mais significativa do número. A instrução equivalente no sistema Z-80 é:

ADD HL, BC (ou o correspondente par de registos)

As instruções que se seguem adicionam dois números de 16 bits (dois bytes em QUANT1 e QUANT2, deixando o resultado nos registos H e L. A flag de transporte é passada a 1 se o bit mais significativo produz o transporte de uma unidade.

8080

```
LHLD QUANT1
XCHG
LHLD QUANT2
DAD D
```

Z-80

```
LD BC, (QUANT1)
LD HL, (QUANT2)
ADD HL, BC
```

Se não possuir nenhum destes sistemas, ou quiser utilizar operandos superiores a dois bytes, será obrigado a utilizar a versão «transporte» das instruções de adição

e subtracção. O exemplo seguinte mostra como funciona a adição e a subtracção com transporte a partir de duas quantidades de 16 bits.

Adição

```
(byte mais significativo)          (byte menos significativo)

   00011100                         01110111        7 287
 + 01011111                         11111100     + 24 572
 +        1 flag de                 ---------    --------
 ----------  transporte             01110011       31 859
   01111100
```

Subtracção

```
   00000100                         00000000         1024
 — 00000000                         10000000     — (+ 128)
 —        1 flag de                 ---------    --------
 ----------  transporte             100000000    +    896
   00000011
```

A primeira adição ou subtracção deve ser realizada sem transporte (excepto no 6502, em que a flag de transporte deve ser colocada em 0 ou 1).

A sequência de instruções que se segue realiza COX + BOX = FOX em quantidades de dois bytes. Considera-se que todos os resultados podem ser guardados em dois bytes. A fim de obter acesso ao byte de menor ordem dos operandos, utiliza-se o endereço do tipo «COX + 1». O byte de maior ordem encontrar-se-á em COX e o de menor ordem estará no endereço seguinte, que pode ser indicado por COX + 1; por exemplo, se o número fosse 256:

COX	COX + 1
00000001	00000000

Em código-máquina tratar-se-á de dois endereços contíguos, por exemplo 300 e 301.

8080

```
LDA COX + 1
MOV B, A
LDA BOX + 1
ADD B
STA FOX + 1
LDA COX
MOV B, A
LDA BOX
ADC B
STA FOX
```

6800

```
LDAB COX + 1
LDAA COX
ADDB BOX + 1
ADDB BOX + 1
ADCA BOX
STAA FOX
STAB FOX + 1
```

6502

```
CLC
LDA COX + 1
ADC BOX + 1
STA FOX + 1
LDA COX
ADC BOX
STA FOX
```

Z-80

```
LD A, (COX + 1)
LD B, A
LD A, (BOX + 1)
ADD A, B
LD (FOX + 1), A
LD A, (COX)
LD B, A
LD A, (BOX)
ADC A, B
LD (FOX), A
```

Todos os sistemas excepto o 8080 possuem uma flag de «overflow», que é passada a 1 quando surge um número com sinal, de nove bits, como resultado da *adição de dois números de um byte com o mesmo sinal.* Esta condição verifica-se quando se usam números negativos na gama —129 a —256, e positivos na gama 128 a 254. Constitui normalmente uma condição de erro.

Até aqui considerou-se que estamos a adicionar e subtrair números inteiros. Pode-se considerar que a vírgula binária se encontra em qualquer ponto de um número simples ou de multi-precisão. Por exemplo, 00011110 pode representar 7,5; neste caso considera-se que a vírgula se encontra antes dos dois últimos bits do byte. As fracções binárias sucedem-se em potências de dois: 0,1 = 0,5; 0,01 = 0,25; 0,001 = 0,125, etc. É necessário descobrir quantas são as fracções binárias de que o tratamento necessita, descontando o espaço necessário para as multiplicações e divisões, que respectivamente aumentam e diminuem o número de algarismos significativos do resultado. É melhor dividir toda a quantidade de entrada pela potência apropriada de 2, de modo a ficar toda numa forma fraccional. Esta potência é designada em técnica de computadores por «*scaling factor*». Será necessário fazer ajustamentos de cada vez que se usa uma subrotina de multiplicação ou divisão, se se pretende manter o «escalonamento» (*scaling*) inicial, realizando finalmente o ajustamento necessário dos números originais na saída. Esta «caça à vírgula binária» constitui um trabalho cansativo e convém ser evitada. Foram concebidas rotinas de vírgula flutuante, que resolvem automaticamente este problema, para a maior parte dos sistemas, devendo ser usadas se se está a trabalhar com números fraccionários. Alternativamente, pode-se utilizar a forma de representação dos números em «decimal de codificação binária» (BCD), caso em que não é necessário preocuparmo-nos com a falta de correspondência entre a vírgula decimal e a vírgula binária. Esta alternativa bastante útil ao trabalho em binário puro será discutida em seguida.

Adição e subtracção de decimais em codificação binária

A representação BCD apresenta cada dígito de um número decimal sob a forma do seu equivalente binário em quatro bits variando entre 0 (0000) e 9 (1001). Num único byte é possível guardar dois valores BCD. Vejamos

alguns exemplos destas formas de representação dos números:

```
 5 = 0000 0101 (0 5)
17 = 0001 0111 (1 7)
42 = 0100 0010 (4 2)
99 = 1001 1001 (9 9)
```

99 é o maior valor que pode ser guardado num único byte, mas é possível tratar números maiores realizando adições e subtracções de multi-precisão. A vírgula decimal pode ser usada sempre que se quiser.

O método BCD é relativamente ineficaz para guardar dados, dado que são usados mais bits do que na representação binária correspondente. É no entanto útil para a conversão do formato de entrada dos caracteres, no caso de se estar a realizar uma interface para instrumentos como voltímetros digitais, ou se se trabalha com números fraccionais.

A adição e a subtracção fornecem automaticamente resultados BCD a partir de números BCD quando se emprega a instrução de «ajustamento decimal» (DAA) depois de cada adição ou subtracção nos sistemas 8080, 6800 e Z-80. Isto afecta a «flag auxiliar de transporte» ou flag de «semi-transporte»: AC no 8080, H no 6800, H (e N se se trata de uma instrução de subtracção) no Z-80. Podem-se usar todas as instruções de adição e subtracção anteriormente descritas, com ou sem transporte. As formas com transporte serão usadas se se deseja trabalhar com mais do que dois dígitos BCD.

No 6502 a flag decimal é passada a 1 pela instrução SED e passada a zero pela instrução CLD. Depois de se encontrar ao valor 1, todas as adições e subtracções subsequentes são realizadas em BCD até a flag ser passada a zero.

A sequência de instruções seguintes realiza as operações PRICE1 + PRICE2 — DISCT nos quatro sistemas. Considera-se que nenhuma quantidade será superior a 99.

8080

```
LDA PRICE1
MOV B, A
LXI H, DISCT
LDA PRICE2
ADD B
DAA
SUB M
DAA
```

6800

```
LDAA PRICE1
ADDA PRICE2
DAA
SUBA DISCT
DAA
```

6502

```
CLC
SED
LDA PRICE1
ADC PRICE2
SEC
SBC DISCT
```

Z-80

```
LD A, (PRICE1)
LD B, A
LD HL, DISCT
LD A, (PRICE2)
ADD A, B
DAA
SUB (HL)
DAA
```

9

MAIS POSSIBILIDADES DA LINGUAGEM ASSEMBLER E DO CÓDIGO-MÁQUINA

Até agora só discutimos programas sem ramificações nem ciclos fechados. Este capítulo descreve o método usado em linguagem assembler para realizar estes dois tipos de tratamento e introduz algumas outras possibilidades interessantes. Os códigos apropriados para funcionamento em código-máquina podem ser descobertos no manual.

«Branching»

Esta técnica permite-nos passar a uma dada parte de um programa no caso de ser verdadeira uma certa condição de uma flag (produzida por uma instrução aritmética ou de comparação), continuando a máquina a obedecer às instruções pela ordem normal no caso de tal condição ser falsa. A curta sequência de instruções apresentada em seguida para o 8080 continua a realização do programa no caso de a adição dar um resultado positivo (usando a instrução JP para saltar ao rótulo NEXT) ou pára (usando a instrução HLT) se esse resultado não é positivo.

```
ADD B
JP NEXT
HLT
NEXT
(resto do programa)
```

As instruções de «branching» do 6800 e do 6502 trabalham apenas com *endereços relativos*, podendo esta forma ser também usada no Z-80. Nesta forma de endereçamento, o segundo byte da instrução apresenta o número de bytes (o número pode ser positivo ou negativo) que devem ser adicionados ao contador do programa a fim de descobrir o endereço para o qual o programa deve «fazer agulha» («branch») se a condição verificada for verdadeira. Usando a «verificação de bit positivo» (BPL) do 6800, a instrução:

400 BPL 4

passaria para a instrução presente no endereço 406 se a condição fosse verdadeira. Dado que a instrução de branching tem no 6800 e no 6502 o comprimento de dois bytes, o contador de programas conteria o endereço 402 quando a instrução branching estivesse a ser obedecida; o valor 4 seria somado a isto — e daí o endereço de 406. A regra consiste em somar (2 + número no operando imediato) para calcular o endereço efectivo. O operando imediato deve estar na gama — 128 a + 127, dado que deve estar contido num único byte. Isto significa que o endereço para onde o programa «faz agulha» se a condição é verdadeira deve encontrar-se a uma distância de — 126 a + 129 bytes da instrução de branching.

Muitos programas assembler permitem-nos colocar um rótulo *(label)* depois da instrução de branching a fim de especificar o endereço para onde se deseja passar, por exemplo

BPL NEXT

Esta forma poderá ser encontrada em muitas listagens de programas publicados para o 6800 e 6502. Liberta-nos do tédio de definir o operando de uma instrução de branching, dado que isto é feito pelo programa assembler. Se estamos a trabalhar directamente em código-máquina,

no entanto, seremos obrigados a inserir um número positivo ou negativo como operando imediato.
Em seguida apresentamos algumas instruções de branching muito úteis nos quatro sistemas que estamos a estudar.

8080	*6800*	*6502*	*Z-80*	
JC	BCS	BCS	JP C, endereço (ou JR C, i)	«Branch» se transporte = 1
JNC	BCC	BCC	JP NC, (ou JR NC, i)	«Branch» se não houver transporte
JM	BMI	BMI	JP M,	«Branch» se flag de sinal = 1
JP	BPL	BPL	JP P,	«Branch» se flag de sinal = 0
JZ	BEQ	BEQ	JP Z, (ou JR Z, i)	«Branch» se flag zero = 1
JNZ	BNE	BNE	JP NZ, (ou JR NZ, i)	«Branch» se flag zero = 0

As formas JR das instruções Z-80 utilizam endereços relativos (i), tal como 6800. O 6800 possui ainda as instruções:

BLT	«Branch» se sinal + overflow = 1
BLE	«Branch» se zero ou sinal + overflow = 1
BLS	«Branch» se zero ou transporte = 1
BGE	«Branch» de sinal + overflow = 0
BGT	«Branch» se zero e sinal + overflow = 0
BHI	«Branch» se «branch» e transporte = 0

Se se estão a usar valores com sinal, BLT, BLE, BGE, BEQ e BNE serão as instruçções 6800 apropriadas. Se se usam valores de dois bytes sem sinal, as instruções apropriadas serão BCS, BLS, BCC, BHI, BEQ e BNE.

Em todos os sistemas existe uma instrução de «branching» *incondicional*. Esta passa para o ponto do programa indicado, e obedece às instruções por ordem a partir desse ponto. Os programas que passam de um ponto para o outro são difíceis de corrigir e de compre-

ender, pelo que esta instrução deve ser usada sem excessos — ou não ser mesmo usada. A forma é a seguinte:

8080 JMP endereço
6800 BRA endereço relativo ou JMP endereço
6502 JMP endereço ou JMP endereço relativo
Z-80 JP endereço

A instrução JMP do 6800 permite-nos ultrapassar qualquer limitação imposta pela gama de endereços relativos que podem ser guardados num único byte. Pode-se usar qualquer das instruções de «branching» do 6800 para, recorrendo a um endereço relativo, nos fazer saltar para uma instrução JMP, que nos pode então tirar da gama habitual. Se se deseja passar para outro ponto do programa como resultado de uma comparação entre dois operandos, pode-se subtraí-los e utilizar em seguida a instrução «branching» para seguir o trajecto apropriado. A sequência de instruções seguinte passa para o rótulo BIG se o conteúdo do DOG > CAT, passa para LIKE se forem iguais e continua sequencialmente se CAT > DOG.

8080	*6800*	*6502*	*Z-80*
LDA CAT	LDAA DOG	SEC	LD A, (CAT)
MOV B, A	SUBA CAT	LDA DOG	LD B, A
LDA DOG	BEQ LIKE	SBC CAT	LD A, (DOG)
SUB B	BPL BIG	BEQ LIKE	SUB B
JZ LIKE		BPL BIG	JP Z, LIKE
JP BIG			JP P, BIG

O uso da instrução «comparar» permite-nos fazer camparações sem uma subtracção, e posiciona as flags apropriadas de tal modo que se possa utilizar uma instrução «branching» conforme os valores dessas «flags». As formas desta instrução são as seguintes (todas elas posicionam as flags apropriadas):

8080

CMP M compara o acumulador com o conteúdo de um endereço especificado (como se compreende lendo o último capítulo) pelo conteúdo dos registos H e L.
CMP R e CPI comparam o acumulador com um registo e um operando imediato, respectivamente.

6800

CMPA e CMPB comparam o conteúdo de um endereço com o acumulador e o registo B respectivamente.
CBA compara o acumulador e o registo B.

6502

O conteúdo de um endereço pode ser comparado com o acumulador, o registo X e o registo Y respectivamente pelas instruções CMP, CPX e CPY.

Z-80

CP (HL), CP R e CP *n* correspondem às três instruções 8080.

Em seguida apresentamos a sequência de instruções 6800 para «branching» em função dos resultados da comparação entre DOG e CAT, re-escrita usando a instrução de comparação.

```
LDAA DOG
CMPA CAT
BEQ LIKE
BPL BIG
```

Ciclos

A maior parte dos programas contêm ciclos, como se explicou no capítulo 1. Já conhecemos a maneira normalizada de programar um ciclo em que o número de elementos termina por uma «sentinela», por exemplo — 1.

Pode-se usar a subtracção ou a instrução de comparação para verificar a igualdade entre cada elemento e a sentinela.

Pode-se igualmente programar um ciclo, se este contiver um número fixo de passos, realizando uma contagem e verificando-a de cada vez que o ciclo é passado. Isto pode ter a desvantagem de utilizar o acumulador, obrigando a armazenar e re-armazenar quaisquer outros totais de cada vez que se aumenta ou verifica a contagem.

Os quatro sistemas em estudo possuem instruções úteis para aumentar ou diminuir de uma unidade um contador num registo ou numa posição de memória, e para posicionar as «flags» apropriadas. Estas instruções são úteis para o tratamento de *arrays* (ver capítulo 1) e de contadores. As formas apropriadas são indicadas em seguida.

8080

INR M e DCR M: incrementa/decrementa de 1 um contador numa posição de memória definida pelos registos H e L.

INR R e DCR R: como acima, para o caso de um registo.

6800

INCA, DECA, INCB, DECB, INC, DEC, INX e DEX realizam as operações citadas para o acumulador, o registo B, a posição de memória e o registo X respectivamente.

6502

As operações citadas são realizadas numa posição de memória e nos registos X e Y por INC, DEC, INX, DEX, INY e DEY respectivamente.

Z-80

INC (HL), DEC (HL), INC R e DEC R correspondem às instruções 8080 referidas.

O programa seguinte utiliza um ciclo para multiplicar o conteúdo de ITEM por cinco (de uma maneira relativamente ineficaz, se bem que mostre o uso dos ciclos). É feita uma comparação de um contador a fim de verificar se não é zero depois de cada diminuição. Se não o for, o ciclo é obedecido mais uma vez. O acumulador é colocado ao valor zero no início do programa, e a multiplicação é realizada adicionando ITEM cinco vezes.

Seria possível realizar contagens alternativas (por exemplo uma negativa) ou realizar verificações alternativas. É sempre necessário assegurar que a verificação afirmativa da condição nos conduz novamente para o início do ciclo, e que quando a condição deixa de ser verdadeira se continua imediatamente o resto do programa. Isto poupa saltos incondicionais desnecessários. Se estiver a programar o 6800 ou o 6502 em código-máquina, portanto, coloque uma quantidade negativa no endereço imediato da instrução de verificação, a fim de saltar para trás para o início do ciclo.

8080

```
   MVI A, 0
   MVI B, 5
   LXI H, ITEM
ST ADD M
   DCR B
   JNZ ST
```

6800

```
   CLRA
   LDX # 5
ST ADDA ITEM
   DEX
   BGT ST
```

6502

```
   LDA # 0
   LDX # 5
ST CLC
   ADC ITEM
   DEX
   BNE ST
```

Z-80

```
   LD A, 0
   LD B, 5
   LD HL, ITEM
ST ADD A, (HL)
   DEC B
   JP NZ ST
```

O início do ciclo é identificado por ST, considerando-se que o produto pode ser guardado num único byte.

O 8080 e o Z-80 possuem ainda instruções de incremento e decremento que afectam um valor de dois bytes num par de registos, mas estas *não* posicionam as flags.

Arrays e tabelas

A ideia de um *array* ou tabela, onde cada elemento é identificado pelo nome da tabela e um índice que dá a conhecer a sua posição no array (por exemplo, A_1, V_{18}) foi já descrita no Capítulo 1. Para traduzir fluxogramas usando arrays em código-máquina ou linguagem assembler é necessário utilizar uma técnica conhecida pelo nome de *endereçamento indexado.* Este adiciona o conteúdo de um registo de índice ao conteúdo do endereço de certas instruções, de tal modo que aumentando (ou diminuindo) o valor do registo de índice pode-se passar por um ciclo que enderece de cada vez um elemento diferente de um array ou tabela. O 6800, o 6502 e o Z-80 possuem registos de índice conhecidos respectivamente por X, X e Y, IX e IY. Todos excepto os do 6502 têm um comprimento de 16 bits. No 6502, o registo de índice Y só pode ser usado com endereços da página 0 (0—255), com as instruções LDX e STX.

As formas de uma instrução típica que use um registo de índice são:

```
6800  LDAA 2, X
6502  LDA TABELA, X
Z-80  LD(IX—2), A
```

O elemento anterior à vírgula nos casos do 6800 e do Z-80 é conhecido pelo nome de *deslocamento.*

O 8080 não possui verdadeiros registos de índice. As instruções de incremento e decremento que trabalham com pares de registos (tais como o útil H, L) não afectam «flags». Se tivermos bastante cuidado e conhecermos os endereços em código-máquina (que, em linguagem assembler, podem ser posicionados por ORG, instrução descrita anteriormente) pode-se utilizar INR L para

diminuir ou aumentar o registo L se se tem a certeza de que não produz um transporte. Alternativamente, pode-se estabelecer um contador separado num único registo, como por exemplo B, e usar então a INX H ou DCX H para aumentar ou diminuir o par de registos H e L. No Z-80 podem utilizar-se técnicas semelhantes.

A sequência de instruções seguinte soma um *array* ou tabela de 12 elementos no endereço inicial PESO no acumulador. Considera-se que o total pode ser guardado num único byte.

8080

```
   LXI, H, PESO
   MVI D, 12
   MVI A, 0
ST ADD M
   INX H
   DCR D
   JNZ ST
```

6800

```
   LDX # PESO
   CLRA
   LDA B # —12
ST ADDA X
   INX
   INC B
   BMI ST
```

6502

```
   LDX # —12
   LDA # 0
ST CLC
   ADC TAB + 12, X
   INX
   BMI ST
```

Z-80

```
   LD HL, PESO
   LD D, 12
   LD A, 0
ST ADD A, (HL)
   INC HL
   DEC D
   JP NZ, ST
```

Subrotinas

As possibilidade de escrever sequências de instruções muito usadas uma só vez num programa e de as chamar novamente tantas vezes quantas se quiser foi já discutida no capítulo 1. Muitas subrotinas úteis dos quatro sistemas estudados foram concebidas para funções como a aritmética de multi-precisão e a aritmética de vírgula flutuante, assim como para funções de entrada e saída.

No 8080 as subrotinas são invocadas pela instrução CALL. Também existem algumas formas condicionais desta instrução. CC, CM, CNC, CNZ, CP e CZ chamam as respectivas condições de transporte, menos, sem transporte, sem zero, positivo, e zero. A instrução RET transfere o controlo para a instrução seguinte a CALL, etc, no programa que chamou a subrotina. Existem formas condicionais de RET (RC, RM, etc.) correspondentes às formas condicionais das instruções CALL e JMP.

O 6800 possui duas instruções de chamamento: BSR (que deve ser seguida de um endereço *relativo* e portanto tem uma aplicação limitada, como se descreveu anteriormente) e JSR (que utiliza um endereço normal ou indexado). O retorno de uma subrotina é realizado através de uma instrução RTS.

O 6502 utiliza as instruções JSR e RTS.

As instruções CALL e as suas formas condicionais CALL NZ, etc., do Z-80, correspondem às instruções 8080, tal como RET, RET NC, etc. A sequência de instruções seguintes utiliza uma subrotina duas vezes para multiplicar uma dada quantidade por três.

8080

```
LXI H, COX
LDA BOX
ADD M
CALL TRIPLE
STA FOX
LXI H, DOG
LDA CAT
ADD M
CALL TRIPLE
STA HOG
TRIPLE MOV B, A
       ADD A
       ADD A
       RET
```

6800

```
LDAA COX
ADDA BOX
JSR TRIPLE
STAA FOX
LDAA DOG
ADDA CAT
JST TRIPLE
STAA HOG

TRIPLE TAB
       ABA
       ABA
       RTS
```


6502

```
LDA COX
CLC
ADC BOX
JSR TRIPLE
STA FOX
LDA DOG
CLC
ADC CAT
JSR TRIPLE
STA HOG
TRIPLE STA TEMP
       CLC
       ADC TEMP
       CLC
       ADC TEMP
       RTS
```

Z-80

```
LD HL, COX
LD A, (BOX)
ADD A, (HL)
CALL TRIPLE
LD (FOX), A
LD HL, DOG
LD A, (CAT)
ADD A, (HL)
CALL TRIPLE
LD (HOG), A
TRIPLE LD B, A
       ADD A, A
       RET
```

A subrotina TRIPLE obriga a que o elemento a triplicar seja colocado no acumulador, e devolve o produto no mesmo local. Excepto no caso do 6502, o conteúdo do registo B é escrito de novo. As subrotinas publicadas indicam sempre quaisquer pormenores quanto a registos ou flags alterados. Se você está a escrever uma subrotina de aplicação bastante geral, convém salvaguardar registos e flags vitais ao entrar na subrotina e restaurá-los antes de sair desta. Na secção seguinte descreve-se um método de conseguir isto.

Operações de empilhamento

Um *stack* (pilha) é uma área de memória onde os dados são guardados e retirados segundo o princípio «o último a entrar é o primeiro a sair» (LIFO — Last In, First Out). Diz-se que o último elemento acrescentado ao stack se encontra *no topo* deste. As instruções têm acesso ao elemento do topo do stack. O stack do 6502 é fixo e

ocupa os endereços entre 100 e 1FF (hexadecimal), num total de 256 bytes. Nos outros sistemas o programador fixa a área do stack.

Todos os sistemas possuem um *apontador de stack* (SP, stack pointer) que aponta a *última* posição do stack usada, isto é, o topo do stack. O apontador do 8080 e do Z-80 deve ser inicialmente carregado pelo programador com um endereço superior em uma unidade à primeira posição do stack, dado que o apontador é sempre diminuído pelas instruções que armazenam dados no stack. As instruções apropriadas são as seguintes.

8080

LXI SP, endereço (por exemplo LXI SP, 4001) carrega um valor no apontador de stack. SPHL transfere o conteúdo dos registos H e L para o apontador de stack.

6800

Ao contrário do 8080, o apontador de stack indica de facto o topo do stack; LDS é a instrução apropriada.

6502

O apontador de stack é carregado do registo X com a instrução TXS.

Z-80

LD SP, endereço, e LD SP, HL correspondem às instruções do 8080.

O stack é principalmente usado para:

— Armazenamento automático do contador do programa (dando o ponto do programa ao qual se deve voltar) quando é chamada uma subrotina. A instrução de chamada diminui automaticamente o apontador de stack de 2 (dado que são necessários dois bytes para guardar o contador do programa) e aumenta-a de 2 quando se abandona a subrotina.

— Armazenamento dos registos e flags ao entrar numa subrotina. É necessário que você mesmo realize estas acções se deseja utilizar o stack deste modo. As instruções apropriadas são:

8080

O acumulador, as flags (chamadas PSW) e os pares de registos apropriados podem ser «empurrados» ou «puxados» do stack (e feito o ajustamento apropriado no contador de programa) através de PUSH A, PUSH PSW, PUSH B, PUSH D, PUSH H, POP A, POP PSW, POP B e POP D.

INX SP e DCX SP aumentam e diminuem SP de um.

6800

PSHA, PSHB, PULA e PULB realizam as mesmas acções com os registos A e B.

6502

O acumulador e as flags podem ser «empurrados» ou retirados do stack e o contador de programa convenientemente alterado através das instruções PHA, PHP, PLA, PLP.

Z-80

PUSH AF e POP AF afectam simultaneamente o acumulador e as flags. Outras instruções, que correspondem às do 8080, são PUSH BC, PUSH D, PUSH HL, POP BC, POP DE, POP HL, INC SP e DEC SP.

A cada «empurrão» para o stack deve corresponder o conveniente «puxão» deste; se se está a usar o stack para armazenamento temporário e se possui mais do que uma saída da subrotina, é necessário assegurar que o apontador de stack esteja no ponto apropriado para cada instrução RET ou equivalente. Uma observação na sua folha de codificação, e a indentação de cada nível de entrada ou saída do stack ser-lhe-ão aqui muito úteis.

Manipulação de bits

Por vezes deseja-se reconhecer os bits individuais no interior de um byte, por exemplo quando se está a dar valores e a testar as flags e se deseja poupar espaço usando palavras de oito flags. Os sistemas que estamos a considerar possuem instruções de «deslocamento» *(shift)* e «lógicas» que o auxiliam neste tipo de programação. Os deslocamentos são também úteis na multiplicação, e por vezes na divisão. Iremos discutir em seguida estas instruções.

8080

Todos os deslocamentos são rotativos, isto é, cada bit que sai por uma das extremidades da palavra reaparece na outra.

	Antes		*Depois*	
	Tr.	*Acum.*	*Tr.*	*Acum.*
RLC roda o acumulador uma posição para a esquerda. O bit mais significativo vai para transporte e para a extremidade da direita. (shift à esquerda).	0	10101010	1	01010101
RRC é o deslocamento correspondente para a direita. (shift à direita).	0	10101010	0	01010101
RAL trata o acumulador e o transporte como nove bits e roda-os nessas condições.	0	10101010	1	01010100
RAR é o correspondente deslocamento para a direita.	0	10101010	0	01010101

6800

	Antes		*Depois*	
	Tr.	*Acum.*	*Tr.*	*Acum.*
ROLA, ROLB, ROL deslocam o acumulador, o registo B ou uma posição de memória da mesma maneira que a RAL 8080.				
ASLA, ASLB, ASL deslocam «aritmeticamente» para a esquerda. O transporte recebe o bit de sinal e entra um zero na extremidade da direita. (shifts aritméticos à esquerda).	1	10101010	1	01010100
LSRA, LSRB, LSR são deslocamentos «lógicos» para a direita. O bit de sinal torna-se 0 e o último bit significativo entra no transporte. (shifts lógicos à direita).	1	10101010	0	01010101

6502

O acumulador ou uma posição de memória podem ser deslocados em rotação para a esquerda, aritmeticamente para o mesmo lado ou logicamente para a direita pelas instruções ROL, ASL e LSR respectivamente. Estas actuam como as correspondentes instruções 6800.

Z-80

RLCA, RRCA, RLA e RRA actuam sobre o acumulador, e correspondem aos deslocamentos 8080 descritos acima. Todos os deslocamentos descritos em seguida actuam sobre

um registo, ou sobre uma posição de memória a que se tem acesso através de um registo de índice ou pelos registos H e L, por exemplo RLCD, RLC(HL).

Em seguida observam-se deslocamentos rotativos como os já discutidos: RL e RR realizam deslocamentos de nove bits (operando e transporte), enquanto RLC e RRC trabalham sobre oito bits de uma maneira análoga às instruções 8080 com o mesmo nome.

SRL e SLA são deslocamentos «lógicos» (apesar do facto de o segundo ser designado aritmético): os bits que saem do operando são perdidos (excepto os que actuam sobre a flag de transporte), e os bits vagos são ocupados com zeros. A acção é a seguinte:

	Antes		*Depois*	
	Operando	*Tr.*	*Operando*	*Tr.*
SLR desloca para a direita	01010011	0	00101001	1
	Tr.	*Operando*	*Tr.*	*Operando*
SLA desloca para a esquerda.	1	01010011	0	10100110
	Operando	*Tr.*	*Operando*	*Tr.*
SRA é um deslocamento aritmético dado que o bit de sinal é repetido, como se mostra nestes dois exemplos. O último bit significativo passa a transporte.	01010101	0	00101010	1
	10101010	1	11010101	0

Os deslocamentos rotativos são usados principalmente para alinhar bits numa palavra, juntamente com as instruções AND e OR descritas na secção seguinte. Como se vê, a combinação dos vários tipos de instrução permite apagar bits de um operando e «empurrar» para o seu lugar bits de outra palavra.

Os deslocamentos lógicos podem ser usados para o mesmo fim. Também multiplicam por dois (deslocamento

para a esquerda) e dividem por dois (deslocamento para a direita), sendo úteis para a construção de subrotinas com estes fins.

O deslocamento aritmético não deve ser usado para alinhar bits porque, se o dígito de sinal do operando original fosse 1, este seria repetido para cada deslocamento aritmético para a direita. Divide o número (consistindo em sete bits e no sinal) por dois, e guarda o quociente.

O Z-80 possui igualmente dois deslocamentos, RLD e RRD, que trabalham sobre uma posição de memória (definida pelos registos H e L) e na metade inferior do acumulador; os quatro bits superiores do acumulador não são afectados. RLD e RRD consideram estes 12 bits como consistindo em três dígitos decimais em codificação binária (BCD). A acção é ilustrada a seguir.

	Antes		*Depois*	
	Acumul.	*Posição*	*Acumul.*	*Posição*
RLD	12	34	13	42
RRD	12	34	14	23

Operações lógicas

Se se deseja poupar parte de um byte (para uma flag ou um número) e guardar qualquer coisa vinda de outro byte ou registo na parte restante, os deslocamentos ajudar-nos-ão a posicionar a parte preservada do byte. As instruções lógicas ajudam-nos a apagar os bits que não interessam e a colocar a informação desejada no espaço deixado livre.

Existem três tipos de instrução lógica. A «aritmética lógica» que realizam num bit 0 e num bit 1, e o efeito que tem o seu uso sobre a totalidade do byte, são apresentados em seguida.

AND ($\wedge$)

```
      0        0        1        1         11100010
AND   0   AND  1   AND  0   AND  1   AND   01010101
-----     -----    -----    -----    --------------
      0        0        0        1         01000000
```

Esta instrução é principalmente usada para eliminar bits não desejados.

OR (OR inclusivo) ($\vee$)

```
     0       0       1       1              11100010
OR   0  OR   1  OR   0  OR   1          OR  01010101
-----   -----   -----   -----          -------------
     0       1       1       1              11110111
```

Esta instrução é principalmente usada para introduzir bits num byte.

OR (OR exclusivo, XOR) ($\oplus$)

```
      0         0         1         1          11100010
XOR   0   XOR   1   XOR   0   XOR   1   XOR    01010101
------    ------    ------    ------    ---------------
      0         1         1         0          10110111
```

Esta instrução raramente é usada. Se o conteúdo de um registo ou de um byte for objecto desta operação em relação a si próprio, o byte do registo é limpo, isto é:

```
     10101010
XOR  10101010
     --------
     00000000
```

É esta a maneira mais rápida de limpar um acumulador ou um registo.

As formas das três operações lógicas para os quatro sistemas considerados são as seguintes.

8080

ANA R, ANA M, ANA I realizam a operação «AND» no acumulador e no operando especificado (R é um registo, M é uma posição de memória definida pelos registos H e L, e I é um operando imediato no segundo byte da instrução).

ORA M, ORA R, ORA I, XRA M, XRA R e XRA I são as instruções correspondentes para as outras funções lógicas.

6800

As funções lógicas são realizadas sobre o conteúdo do acumulador ou registo B pelas instruções ANDA, ANDB, ORAA, ORAB, EORA, EORB. O operando pode ser uma posição de memória, um endereço indexado ou um operando imediato.

6502

As funções lógicas são realizadas sobre o conteúdo de uma posição de memória ou operando imediato e o conteúdo do acumulador por AND, ORA, EOR.

Z-80

AND R, AND (HL), AND N, OR R, OR (HL), OR N, XOR R, XOR (HL) e XOR N correspondem às instruções 8080 indicadas acima.

Representação de caracteres

Se se pode obter um carácter directamente a partir de um dispositivo de entrada antes de ser processado, é provável que obedeça ao código ASCII (American Stan-

dard Code for Information Interchange), que representa todos os números, letras e símbolos especiais em sete bits. Por vezes existe um oitavo bit funcionando como bit de *paridade*, a fim de indicar se o byte foi alterado em transmissão. Por exemplo, o bit de paridade pode ser passado ao nível um ou zero, antes da transmissão, a fim de tornar *par* o número total de uns no byte. Em seguida, ao ser recebido, contam-se os números de uns: se não é par a transmissão foi incorrecta.

Conversão de código para números constitui um útil exemplo das instruções de deslocamento e lógicas já descritas. Os caracteres podem ser comparados utilizando as instruções de comparação e de branching se se pretende realizar o processamento de um texto. *Não* se deve, no entanto, tentar realizar operações aritméticas nas formas ASCII dos números.

Os números ASCII correspondem de perto à forma BCD. Se se considera que o oitavo bit (de paridade) de um carácter ASCII armazenado num byte é zero, os equivalentes ASCII e BCD dos dígitos decimais são:

Decimal	*ASCII*	*BCD*
0	00110000	00000000
5	00110101	00000101
9	00111001	00001001

A sequência de instruções que se segue examina um byte designado STATUS para verificar se os caracteres foram lidos correctamente. Se o bit na posição «8» de STATUS (0000X000) for 1, os caracteres não foram lidos correctamente e o programa passa para o rótulo ERRO. Se a leitura está correcta, são inseridos dois caracteres ASCII nos endereços CHARA e CHARB. A sequência de instruções armazena-os num byte designado NUMERO em forma BCD. A palavra STATUS (formada por flags) não deve ser destruída. O B no operando imediato a seguir às instruções lógicas implica que este operando se encontra em forma binária.

8080

```
LDA STATUS
ANI 00001000B
JNZ ERRO
LDXI H, NUMERO
STA NUMERO
XRA M
STA NUMERO
LDA CHARA
ANI 00001111
RLC
RLC
RLC
RLC
STA NUMERO
LDA CHARB
ANDA 00001111B
ORA M
STA NUMERO
```

6502

```
LDA STATUS
AND # 00001000B
BNE ERRO
LDA NUMERO
EOR NUMERO
STA NUMERO
LDA CHARA
AND # 00001111B
ASL
ASL
ASL
ASL
STA NUMERO
DLA CHARB
AND # 00001111B
ORA NUMERO
STA NUMERO
```

6800

```
LDAA STATUS
ANDA # 00001000B
BNE ERRO
LDAA NUMERO
EORA NUMERO
STAA NUMERO
LDAA CHARA
ANDA # 00001111B
ASLA
ASLA
ASLA
ASLA
STAA NUMERO
LDAA CHARB
ANDA # 00001111B
ORAA NUMERO
STAA NUMERO
```

Z-80

```
LD A, (STATUS)
AND 00001000B
JP NZ ERRO
LD HL, NUMERO
LD, A NUMERO
XOR (HL)
LD (NUMERO), A
LD A, (CHARA)
AND 00001111
RLCA
RLCA
RLCA
RLCA
LD (NUMERO), A
LD A, (CHARB)
AND 00001111
OR (HL)
LD (NUMERO), A
```

Entrada e saída

O simples armazenamento de dois caracteres BCD a partir de duas posições de memória dar-nos-á alguma indicação do que se encontra envolvido na transferência de dados dos e para os dispositivos de entrada e saída *depois* de ter sido colocado numa posição de memória.

O problema acima tratou apenas de um par de dígitos BCD e não teve em conta caracteres errados, caracteres redundantes (por exemplo espaços), números negativos, números binários ou números em vírgula flutuante, ou ainda o processo requerido para transferir um carácter de um dispositivo como um teclado, um leitor de fita ou uma disquete para uma posição de memória.

Os dispositivos de entrada e saída não se encontram de modo algum normalizados, aconselhando-se fortemente o leitor a utilizar as subrotinas de entrada e saída fornecidas para o equipamento que usa ou que possam ser obtidas em fontes publicadas. Se estas não contiverem uma conversão de caracteres para as formas aritméticas (binária, BCD, vírgula flutuante) que deseja usar, existem numerosas subrotinas publicadas para a conversão dos caracteres ASCII para qualquer destas formas. A programação de uma rotina de entrada-saída, particularmente para um dispositivo periférico complexo como uma disquete ou um vídeo, deve ser deixado aos «virtuosos». Muitos munitores contêm sistemas de entrada e saída para os periféricos «standard» usados.

A transferência de dados para e de periféricos lentos como um teclado, um leitor de fita em papel, etc., é feito carácter a carácter. Chama-se a isto *entrada-saída programada*. Seria demasiado lento para periféricos rápidos como os discos e alguns vídeos, em que são realizadas transferências de caracteres em bloco usando uma técnica designada por *acesso directo à memória* (DMA). A transferência do bloco é realizada automaticamente sem qualquer intervenção do programador.

Quer seja usado ou não o DMA, o programa deve realizar a tarefa de converter cada carácter e verificar o

estado do dispositivo a fim de concluir se está «pronto» ou «ocupado», ou se ocorreu alguma deficiência durante a transmissão.

A conversão é realizada antes da transmissão de um carácter de saída e depois da transmissão de um carácter de um dispositivo de entrada. Os espaços, as linhas novas (retornos do carro), as vírgulas decimais e as outras pontuações devem ser tidas em conta.

Um bit da palavra de «status» do dispositivo permite ao programador determinar se o dispositivo está ainda ocupado com a transferência do carácter ou bloco anterior. Se o bit está em zero, o programa pode ser levado a entrar num ciclo até o dispositivo estar pronto para as transferências. Outros bits da palavra de «status» são usados para fornecer informações sobre quaisquer erros que se possam verificar, como seja o erro de paridade ou condição final de fita no caso de um leitor de cassette. Alguns bits realizam ainda uma função de controlo, como seja pôr em funcionamento ou parar um leitor de cassette. Se o «status» é satisfatório, pode-se realizar a transferência de dados por DMA ou por entrada-saída programada.

Como alternativa ao ciclo realizado até o dispositivo estar pronto para uma transferência de dados, o sistema pode ser programado de tal modo que seja o que for que estiver a ser processado verificar-se-á imediatamente uma interrupção deste processamento (usando a possibilidade de *interrupção* descrita na secção seguinte) quando o dispositivo estiver pronto. Isto permite que se realize outro processamento durante os intervalos entre transferências, e é vital se, além de se trabalhar com um dispositivo de entrada-saída, se está igualmente a controlar um dispositivo externo cujo funcionamento não possa ser sempre suspenso durante o tempo necessário para a outra transferência. Se forem possíveis interrupções (não o devem ser durante a parte do programa que é vital para o controlo do dispositivo externo), o programa obedecerá à rotina de entrada-saída sempre que o dispositivo se encontra pronto.

Interrupções

Como se mencionou acima, a possibilidade de interrupção permite interromper automaticamente um programa a fim de atender às necessidades de um dispositivo de entrada-saída. Só uma pequena porção dos 2 milisegundos entre o aparecimento de caracteres sucessivos num leitor de fita de alta velocidade é ocupado pelo processamento do carácter, normalmente cerca de 3 micro-segundos. Pode-se então continuar o programa interrompido até ocorrer uma nova interrupção.

As interrupções são igualmente úteis para forçar a entrada numa secção de programa de modo a corrigir condições *urgentes* (como a comutação de linhas num modelo de caminhos de ferros) ou para apresentar uma mensagem de alarme, como seja a passagem da temperatura máxima requerida num sistema de controlo de aquecimento. Podem também ser usadas juntamente com um relógio, interrompendo automaticamente a intervalos previamente definidos a fim de atender a qualquer outra condição desejada.

Uma flag de interrupção permite que as interrupções sejam reconhecidas ou ignoradas. Todos os sistemas que estamos a estudar, excepto o 8080, possuem uma interrupção «não-mascarável» que não pode ser ignorada e que pode ser usada para funções vitais.

Quando a interrupção ocorre, e a menos que seja decidido ignorá-la, o programa continua para um endereço previamente definido e obedece às instruções a partir desse ponto. O contador de programa é armazenado quando se entra no programa de interrupção e re-armazenado quando a rotina está terminada, tal como no caso da entrada ou saída de uma subrotina. As rotinas de interrupção devem ser escritas numa secção da memória que não possa ser alterada por acidente, e devem ser construídas cuidadosamente no caso de o processamento dever ser realizado num período de tempo muito curto. As possibilidades de interrupção dos quatro sistemas são brevemente discutidas em seguida.

EI permite a entrada de interrupções; DI impede-a. Normalmente, quando se entra numa rotina de interrupção impedem-se outras interrupções até a rotina presente ser terminada. A instrução RET é usada quando se deseja voltar ao programa principal depois de uma rotina de interrupção, como acontece ao programar subrotinas.

Podem existir até oito dispositivos capazes de causarem uma interrupção. Os endereços iniciais das rotinas de interrupção destes dispositivos são 0, 8, 16, 24, 32, 40, 48 e 56. Dado que não é provável que todo o processamento da interrupção possa ser realizado com oito bytes, o endereço de início de uma dada interrupção salta normalmente para uma área da memória que contém a subrotina apropriada. Usa-se um equipamento designado módulo **de interrupção prioritária ou** unidade de controlo de interrupção prioritária quando se pretende definir níveis de prioridade entre os dispositivos que podem provocar uma interrupção. Isto organiza as prioridades no caso de se verificarem pedidos simultâneos de interrupção.

Quando é permitida a interrupção, introduz-se uma instrução RST no *bus* de dados. Esta salvaguarda *apenas* o contador do programa, antes de se entrar na rotina de interrupção num dos endereços acima. Se além disso pretende salvaguardar quaisquer registos e flags na área de *stack* ou de memória, você é obrigado a tomar conta do assunto no início da rotina de interrupção, e em muitos casos também é obrigado a actuar quando pretende accionar a flag de impedimento da interrupção. No final do processamento apropriado deve novamente actuar no sentido de restaurar o conteúdo original dos registos e das flags, accionando a flag que permite nova interrupção, antes de terminar com uma instrução RET. Isto permitirá à máquina voltar ao ponto do programa principal onde se encontrava.

6800

A «máscara» de interrupção (nome dado à flag) é accionada pela instrução SEI e anulada por CLI. SWI é uma instrução que permite uma interrupção pelo programa, em vez de realizada por um dispositivo externo; transfere o controlo para um endereço guardado nas posições de memória hexadecimais FFFA e FFFB, que constituem a primeira instrução da rotina que trata este tipo de interrupção. Tal como em todos os tipos de interrupção 6800 discutidas, o conteúdo do acumulador, do registo B, do registo de índice e das flags, tal como o contador de programa, são preservados no «*stack*» e recuperados no final. A saída da rotina é feita através da instrução RTI.

A interrupção que não pode ser ignorada (NMI) («não-mascarável») só é usada em condições de falta de energia.

A única interrupção permitida a partir de um dispositivo externo vai para as posições de memória FFF8 e FFF9 (hexadecimal) a fim de obter o endereço de início da rotina que trata a interrupção. A preservação dos registos e das flags, assim como o uso da instrução RTI, ocorrem tal como na interrupção de software, SWI. Se se pretendem obter interrupções a partir de mais do que um dispositivo externo, os sinais de interrupção sofrem uma operação lógica OR; em seguida, depois de ocorrer a interrupção, o bit de «status» (de estado) de cada dispositivo é examinado a fim de se verificar qual deles provocou a interrupção.

6502

A flag de interrupção é accionada por SEI e anulada por CLI.

A instrução para o início da rotina que trata a interrupção não-mascarável é armazenada nas posições de memória FFFA e FFFB. Só são armazenadas o contador de programa e as flags. O retorno é realizado através de RTI.

O programa de interrupção externa inicia-se nas posições FFFE e FFFF. Só são armazenadas o contador de programa e as flags. Se se pretende trabalhar com vários dispositivos capazes de interromper é necessário utilizar a técnica descrita acima para o 6800.

Z-80

O Z-80 possui três modos de circuito de interrupção mascarável — definidos pelas instruções IM0, IM1 e IM2 — além de uma interrupção não-mascarável. Quando esta última ocorre o contador de programa é poupado e guardado no *stack*, e o controlo é transferido para o início da rotina que trata a interrupção, que se encontra armazenada na posição de memória 0066 (hexadecimal). As outras interrupções são automaticamente anuladas enquanto esta rotina está a ser obedecida. O retorno é realizado através de uma instrução RETN.

O modo de interrupção 0 assemelha-se à interrupção usada no sistema 8080 e já discutida. O retorno é realizado através de uma instrução RETI.

O modo de interrupção 1 é semelhante à interrupção não-mascarável, excepto no facto de poder ser ignorada no caso de a flag de interrupção ser limpa por uma instrução DI, e o endereço inicial do único programa permitido encontra-se na posição de memória 0038 (hexadecimal). O retorno é realizado através da instrução RETI.

No modo 2 são possíveis 128 rotinas de interrupção. O endereço inicial de cada uma destas encontra-se guardado num endereço constituído pelo byte de maior ordem do *registo vector de interrupção* (que é carregado com LD I, A) e pelo byte de menor ordem fornecido pelo dispositivo que provoca a interrupção. Se o registo do vector de interrupção (ou I) contém FF e o dispositivo fornece 16, encontrar-se-á o endereço da rotina que processa esta interrupção na posição FF16 (hexadecimal).

Estes endereços são normalmente juntos numa tabela de vectores de interrupção, que pode conter 128 entradas. Quando se entra numa rotina de interrupção de modo 2, só é armazenado no *stack* o contador de programa. A saída da rotina é realizada através da instrução RETI.

10

DESENVOLVIMENTO E TESTE DE PROGRAMAS

É improvável que qualquer programa, excepto os mais simples, trabalhe quando é pela primeira vez introduzido na máquina através do teclado ou comutadores. Não deve ficar desiludido com isto. A velocidade normal da indústria americana para programação de microcomputadores em linguagem assembler é de dez instruções por dia completamente testadas. No caso de uma linguagem de alto nível, um dos mais importantes utilizadores de computadores em Inglaterra afirma que a produção média de um programador é de 21 instruções por dia, enquanto um dos principais fabricantes de computadores prevê uma produção diária de 15 instruções.

Você verificará que a documentação reunida sobre cada programa, como se mencionou no final do Capítulo 1, é extremamente importante quando se procura testar um programa. O teste inicia-se pelo fluxograma. Os diversos níveis de fluxograma devem estar relacionados com as instruções empregues qualquer que seja a linguagem de programação. No caso da linguagem assembler ou do código máquina pode ser necessário dispor de um nível no fluxograma que relacione as instruções muito detalhadas com os passos lógicos da tarefa que se pretende realizar. Se o primeiro bloco do seu fluxograma é indicado pela letra A, todas as divisões em passos lógicos e instruções de programa necessários para realizar

a tarefa enunciada nesse bloco devem possuir uma referência inicial A.

Se está a trabalhar num sistema microcomputador comercial ou educacional deve tentar que o seu fluxograma seja verificado por outra pessoa, pois você pode deixar-se absorver de tal modo pela sua criação que fica cego a erros lógicos óbvios.

O tempo perdido no teste de um fluxograma é amplamente pago pela considerável diminuição do tempo que será perdido à frente do teclado testando o verdadeiro programa. Na maior parte dos problemas você conseguirá encontrar dados para os quais já conhece a resposta e que pode utilizar para verificar a solução que o fluxograma deveria achar (só alguns problemas de simulação e de investigação operacional se tornarão mais complicados). Se está a programar um jogo, talvez seja obrigado a verificar o programa jogando de facto com cartas reais ou lançando um dado nos pontos apropriados.

Quando se ensaia um fluxograma é necessário fazer num papel uma lista de todas as variáveis (por exemplo o peso, a contagem e o total no problema discutido no primeiro capítulo) e inserir valores alterados depois de cada passo do fluxograma. Deverá também fazer uma coluna para cada decisão do programa, escrevendo «sim» ou «não» para cada teste (como no exemplo daquele capítulo). Reveja cada passo, alterando valores nas colunas onde for conveniente. Pode necessitar de usar uma calculadora se o processamento for mais complicado.

Alguns outros aspectos a considerar na verificação de fluxogramas serão:

— Utilize pequenas quantidades de dados, como no exemplo do capítulo 1 quando um ciclo que deve ser obedecido 12 vezes é verificado apenas 3 vezes.

— Assegure que cada trajecto do programa (incluindo condições de erro) seja verificado. Uma rotina de impostos sobre rendimentos, por exemplo, deve cobrir todos os tipos de rendimentos assim como o primeiro mês e uma codificação de emergência.

— Se está a testar um programa de actualização de ficheiros, utilize um ficheiro pequeno contendo um exemplo de cada tipo de registo.

— Se está a escrever um programa para uma aplicação comercial, consulte o departamento apropriado para verificar se considerou todas as alternativas e combinações de dados possíveis e todas as condições de erro; a experiência do pessoal que trabalha no departamento em causa é certamente maior que a sua. A verificação do fluxograma revelará provavelmente algumas condições de erro que não considerou anteriormente.

— Verifique se inclui mensagens para o operador relativas ao tipo e ao número de elementos que devem ser comunicados por teclado à máquina, ou o tempo necessário para inserir um novo ficheiro em cassette. Esta precaução é vantajosa mesmo que você seja a única pessoa a utilizar provavelmente o programa, dado que ao fim de algum tempo pode esquecer alguns destes factores.

Fases do teste de um programa codificado

Depois de se ter verificado o rigor lógico do fluxograma —, este será codificado ou traduzido em instrução da linguagem de programação que está a utilizar. Para facilitar o ensaio do programa, e para o auxiliar no caso de o querer alterar depois de estar completamente operacional, é conveniente dividi-lo (se o seu comprimento o justificar) em secções autónomas de não mais de 50 instruções. Chama-se a isto, profissionalmente, *programação modular*. Certas variáveis ou posições serão reservadas para passar dados de um módulo para outro. Por vezes justifica-se escrever todo o programa sob a forma de uma série de subrotinas, de tal modo que o programa principal em Basic consiste inteiramente em instruções GOSUB, e em Fortran em instruções CALL; em código-máquina e em linguagem assembler será constituído principalmente pelas diversas instruções CALL e

JSR. A instrução BREAK, que se encontra em alguns monitores como o Zilog MCB, pode ser usada nos pontos convenientes de um programa em linguagem assembler a fim de parar a execução do programa de modo a permitir o exame de registos e posições de memória apropriados.

Se está a utilizar um programa que não foi escrito por si este poderá necessitar de alguns ajustamentos para poder ser adaptado ao seu sistema. Se houver necessidade de muitas alterações, o programa deverá ser testado tão extensivamente como no caso de ter sido escrito de novo. Qualquer programa Basic escrito para um microprocessador diferente do seu (ou para um minicomputador ou computador grande) deverá ser examinado cuidadosamente; pode utilizar possibilidades (funções e instruções) não existentes na versão de Basic por nós usada, e pode até utilizar algumas abreviações de instruções (por exemplo P em vez de PRINT) que não podem ser transferidas para outro sistema. Em alguns casos você será obrigado a consultar o manual de programação do sistema para o qual o programa foi originalmente escrito ou determinar o significado de uma abreviatura ou o funcionamento exacto de uma função ou instrução estranhas. Os programas Fortran são mais «portáteis» do que o Basic, apesar de muitos fabricantes lhe acrescentarem extensões pouco significativas. Em programas em código-máquina e em linguagem assembler escritos originalmente por outra pessoa, tenha cuidado com os saltos para subrotinas de que o seu sistema não disponha. Se está a utilizar um programa monitor pode descobrir que é obrigado a alterar os endereços de um programa escrito por outrem, por se sobreporem às posições do monitor.

As fases de ensaio de um programa são:

1. Verificação das instruções.
2. Exame do programa procurando erros textuais e «gramaticais», como vírgulas deslocadas, e, em Basic, uma declaração FOR sem a correspondente NEXT.

3. Compilação no caso de não se estar a usar um programa interpretador.
4. Teste de cada módulo do programa com dados cuidadosamente escolhidos a fim de assegurar que todos os ramos do programa sejam obedecidos.
5. Teste do conjunto do programa depois de se assegurar que todos os módulos se encontram livres de erros.
6. No caso de um sistema comercial, funcionamento paralelo do computador e do sistema por ele substituído, a fim de verificar se aquele consegue processar grandes quantidades de dados em condições operacionais. No caso de um sistema de controlo, por exemplo accionamento de um aquecimento ou de um modelo de caminhos de ferro, verificar se o sistema trabalha quando se encontra ligado a todos os dispositivos externos.

Verificação das instruções

Este teste é semelhante ao realizado sobre o fluxograma, excepto no facto de se estar a trabalhar ao nível das instruções de programa, o que no caso do código-máquina ou de uma linguagem assembler será muito mais pormenorizado do que com os blocos do fluxograma. Você deve fazer um registo contínuo do conteúdo das variáveis, registos e stacks, assim como do estado das flags e comutadores. Esta verificação das instruções permitir-lhe-á poupar tempo no computador e frustração frente ao teclado, devendo ainda assegurar que todos os trajectos previstos no programa sejam seguidos.

Eliminação dos erros «gramaticais»

O exame das instruções do programa quanto à correcção da respectiva *forma* permitirá poupar tempo mais tarde. O ensaio realizado antes de apresentar um programa a um sistema microcomputador é particularmente

importante quando se está a utilizar um compilador. Nada é mais frustrante do que realizar duas ou três tentativas de compilar um programa errado devido ao deslocamento de uma vírgula, uma referência a um nome ou um número de linha não existentes, etc.

A verificação do texto é particularmente aborrecida se se está a utilizar o código-máquina e a comunicá-lo ao computador através de comutadores e não de um teclado hexadecimal. Não há então alternativa; é-se obrigado a verificar cuidadosamente cada padrão de uns e de zeros, comparando-os, até se conhecer bem o sistema, ao padrão indicado no manual de programação.

Até estar habituado a qualquer linguagem de programação, convém recorrer constantemente ao manual quando se verificam os possíveis erros de texto. Estes erros surgem com menor probabilidade se usar instruções simples e não tentar realizar muitos passos numa única instrução de nível elevado. Isto, particularmente no caso de funções; pode dar origem ao aparecimento de um parêntesis que não é fechado.

Alguns erros comuns, além dos já mencionados, são os seguintes:

— Pontuação incorrecta;
— Uso incorrecto dos espaços;
— Forma incorrecta da instrução;
— Abertura de um ficheiro mais de uma vez;
— Referência a um número de dispositivo incorrecto;
— Confusão entre o número 0 e a letra O maiúscula;
— Aspas não fechadas.

O passo seguinte, se se está a utilizar um compilador ou assemblador, consiste em compilar ou «assemblar» o seu programa de tal modo que possa executar módulos deste. Durante esta compilação serão indicados todos os erros gramaticais que ainda não tiver corrigido. Se está a usar um interpertador, todos os erros gramaticais serão notados nesta fase, sendo apresentada a mensagem de

erro apropriada quando se escreve RUN no teclado e se tenta executar o programa.

Teste da execução do programa

Finalmente você vai ver o que o programa consegue fazer. Nesta fase a cassette de fita e uma impressora são extremamente úteis. A fita pode guardar uma cópia do programa que se encontra na memória quando se começa a testar o programa. Muitas vezes far-se-ão tantas alterações durante o período de testes que facilmente se esquece aquilo com que se começou. No final de cada sessão de ensaio é também bom fazer uma cópia do estado actual do programa, a menos que se decida que as alterações não produziram uma melhoria clara. Naturalmente devem-se anotar todas as modificações efectuadas, mas uma cópia em fita permitirá guardar alguma de que você se tenha esquecido, dado que podem praticar-se alterações contínuas num dado momento de «genialidade» em que nos encontramos sentados à frente do teclado. A impressão é útil para fornecer uma cópia definitiva do programa no início e no final de cada sessão de ensaio, assim como das alterações realizadas.

Alguns sistemas em «package» detectarão certos erros durante o funcionamento do programa, quando se tenta levar o sistema a realizar uma impossibilidade. Isto dá origem a mensagens de erro como:

— Divisão por zero;
— Ausência de dados;
— Ficheiro não existente;
— «Overflow» (resultado para além do número máximo que pode ser usado).

O sistema parará depois de apresentar a mensagem de erro, cabendo-lhe depois a si descobrir o que deu origem ao erro.

Muitos sistemas possuem bons auxiliares de diagnóstico ligados ao respectivo compilador de Basic ou de outra linguagem de alto nível, ou ao monitor de um programa em linguagem assembler ou em código máquina. Estes podem ser muito úteis na determinação do que aconteceu quando se obteve uma mensagem de erro, ou quando os resultados não têm sentido, ou ainda quando nada acontece durante tanto tempo que somos forçados a concluir que o computador está a executar um ciclo interminável. Um sistema possui diagnósticos Basic que incluem:

— Listagem dos valores das variáveis usadas;
— Listagem dos números de linhas executadas entre linhas declaradas — a fim de mostrar qual o trajecto que a máquina está a seguir, e se entrou inadvertidamente num ciclo sem fim.

Alguns monitores e sistemas de desenvolvimento para programas em linguagem assembler e código-máquina possuem algumas facilidades equivalentes. Um sistema possui:

— Marcação de ponto de paragem *(brekpoint)* — para realizar uma pausa na execução de um programa da primeira vez (ou mais tarde) em que é executada uma dada instrução;
— Apresentação do conteúdo dos registos depois de cada instrução ser obedecida;
— Apresentação de posições de memória escolhidas.

Se o seu sistema não possui estas facilidades para o auxiliar a descobrir os erros dos programas, você deve tentar inserir instruções de impressão no programa a fim de a máquina apresentar as variáveis e o trajecto que executa, no caso dos programas em linguagem de alto nível. Em Basic, pode-se identificar cada ciclo por um número e imprimir este de cada vez que aquele é obedecido. Se está a ensaiar programas a níveis de linguagem inferiores sem o auxílio de facilidades de diagnóstico, deve inserir instruções para imprimir um qualquer símbolo de cada vez que se executa um ciclo. Estas indicações deverão ser retiradas do programa depois de ter sido ensaiado e se encontrar operacional.

Você deve no entanto considerar cuidadosamente, quer esteja ou não a programar numa linguagem de alto nível, a relevância dos elementos que se propõe imprimir ou apresentar no «display»; não sobrecarregue o programa na fase de teste com instruções supérfluas.

Um auxiliar de diagnóstico muito vulgar se não se está a programar numa linguagem de alto nível é o «dump» (esvaziamento) da memória. Torna-se neste caso possível obter, num ponto em que o programa termina a execução, uma impressão (se o sistema possuir uma impressora) do conteúdo da memória (normalmente em hexadecimal). Se está a usar um interpretador Basic, você conseguirá passar o seu programa à razão de uma instrução de cada vez, de modo a verificar qual o trajecto que está a seguir e descobrir o conteúdo das variáveis vitais. Alguns sistemas para programação a um nível inferior (por exemplo o Kim) também possuem esta característica.

Durante o ensaio é provável que você acrescente emendas ao programa. Deve anotar cuidadosamente estas emendas na cópia da codificação que se encontra à sua frente. A possibilidade de parte delas o iludirem no momento em que já está um pouco confuso devido ao grande número de alterações já foi discutida quando se descreveram as vantagens da obtenção de uma cópia do programa no final de uma sessão de ensaio. Um dos poucos usos justificáveis da instrução GO TO em Basic ou noutras linguagens de alto nível consiste na inserção dessas emendas possivelmente temporárias no seu programa — mas quando são de facto úteis devem ser numeradas pela sequência em que ocorrem. Alguns Basics permitem numerar de novo as instruções a fim de facilitar a inserção de alterações convenientes ou necessárias. Se tiver seguido o hábito de deixar 10 números vagos entre cada linha (numerando-as 100, 110, 120, etc.), conseguirá inserir emendas sem dificuldade mesmo que

o seu sistema não permita alterar a numeração. Em linguagem assembler ou código-máquina é conveniente, se dispuser de espaço de memória, deixar uma página de memória livre para as emendas; a página 0 é muitas vezes usada para este fim. Se está a utilizar linguagem assembler utilizará provavelmente muito mais as emendas para evitar a necessidade de refazer frequentemente o programa, em particular se puder colocar a versão em código máquina numa cassette. A instrução NOP (ausência de operação) é útil repetida várias vezes no final de cada secção de um programa, pois permite ser substituída por emendas.

É difícil generalizar os erros que podem levar um programa a falhar nesta fase. Alguns, mais vulgares, serão:

— Erro de entrada: é fácil haver enganos ao usar o teclado, o que pode ter como resultado enviar à máquina um valor que conduzirá a ramais do programa inesperados.

— Os ramais que se seguem a uma comparação são invertidos: por exemplo, codificação que é concebida para um ramal > 12 segue-se ao trajecto alternativo. Isto pode acontecer mais facilmente do que se pensa, particularmente quando se utilizam duas instruções de comparação a seguir uma à outra.

— Uma instrução FOR em Basic (ou o seu equivalente) pode nunca atingir o valor final devido a um erro de entrada, a um engano lógico no processamento, ou não se terem previsto erros de arredondamento ou diferenças de representação em vírgula flutuante dos números decimais (mencionado mais atrás).

— Um endereço para um array que exceda os limites deste.

Alguns erros comuns em programação em código-máquina e em linguagem assembler são:

— Uso da instrução incorrecta entre as muitas alternativas «somar» e «subtrair».

— Re-escrever o conteúdo de um registo.

— Flags com valores diferentes dos esperados.

— Perda do conteúdo de registos e flags quando ocorre uma interrupção.

— Confusão entre os diversos modos de endereçamento indirecto.

— Erros de escalonamento.

— Interferência com secções do programa Basic, se se está a usar a declaração POKE em programação de linguagem mista.

Deve-se pensar um pouco na melhor maneira de ensaiar programas que são accionados por comutadores em ligação com dispositivos exteriores, como um alarme ou um termostato. As secções invocadas pelo comutador devem ser accionadas isoladamente, sendo inseridos dados num endereço a fim de simular o funcionamento de um dispositivo externo. Por vezes, na fase final de teste é possível construir um circuito simples, usando lâmpadas que representem o dispositivo externo.

Quando se ensaiaram os módulos do programa de maneira satisfatória pode-se passar todo o programa. Se este tem um comprimento ou uma complexidade relativamente grandes, você pode verificar surpreendido deficiências quando o programa é passado completamente. Dado que já verificou os módulos, os erros encontrar-se-ão inevitavelmente na passagem dos dados de um módulo para outro. Muitas vezes parte se de princípios errados no cálculo prévio de um dado valor. Esta fase do ensaio não deve porém ser muito demorada.

Se está a ligar o microcomputador de modo a controlar um dispositivo externo, pode em seguida verificar o seu funcionamento ligando-o ao seu alarme, modelo de comboio, etc. Em tratamento de dados comerciais, depois de todo o programa ser ensaiado você verificará normalmente que este pode ser ligado a outros numa sequência; por exemplo, um programa de lista de pagamentos pode ser ligado a programas sobre os impostos a pagar anualmente, totais a pagar a seguros, etc.

Quando se passa pela primeira vez uma destas sequências é habitual fazer funcionar paralelamente o sistema substituído de modo a confirmar a correcção dos programas.

O teste de programas pode parecer trabalhoso, mas é muito estimulante. A necessidade de registar tudo o que se faz não consome tanto tempo como se pensa, e se possui um gravador de cassettes ligado ao sistema não é difícil preservar diversas secções do programa durante o seu ensaio. Você verificará que algumas das deficiências na construção dos primeiros programas depressa serão eliminadas. Talvez descubra também que é melhor fazer programas simples: evite um trabalho excessivo numa única instrução de alto nível, ou «truques», como a subtracção de um valor a si próprio a fim de limpar um registo ou a soma de dois valores binários idênticos a fim de os deslocar de um espaço para a esquerda em instruções em código-máquina ou em linguagem assembler.

Depressa ganhará confiança e verificará que está a desenvolver um «estilo», concentrando-se no uso de apenas uma das muitas maneiras de escrever certas secções de um programa. Acabará por atingir uma fase em que consegue ler um programa numa revista de informática e sentir (e talvez justificadamente) que o seu próprio sistema pode fazer melhor, mais depressa e de maneira mais económica.

A programação é essencialmente um trabalho criativo, que você acabará por achar bastante estimulante — muito mais aberto do que resolver puzzles ou mesmo problemas de xadrez, por exemplo. Excepto no caso de programas triviais, não existem dois programadores diferentes que escrevam programas idênticos para resolver um mesmo problema. Antes do mais, procure ter prazer nesta actividade; espero que tenha a sorte de nunca enfrentar problemas complicados na execução dos seus programas.

SOLUÇÕES SUGERIDAS PARA OS EXERCÍCIOS

Existe um número infinito de soluções correctas para um problema de programação — basta pensar no número de variáveis alternativas que se podem usar apenas em Basic! A ordem das expressões aritméticas, a ordem das declarações e a quantidade de tratamento realizada numa única declaração podem variar consideravelmente, no entanto cada combinação pode dar uma mesma resposta correcta. As declarações REM variarão também com as preferências de cada programador.

As soluções aqui sugeridas são portanto apenas uma das maneiras de resolver os problemas indicados nos exercícios. Todas foram verificadas num microcomputador. Considera-se que a versão de Basic que o leitor está a usar não está limitada a números inteiros, mas nas soluções utilizam-se apenas as declarações e funções comuns às versões mais reduzidas de Basic que não se limitam ao trabalho com números inteiros. Em todos os casos, a utilização de «atalhos» engenhosos, que poderão ocorrer-lhe a si, foi escrupulosamente evitada. Não se fez qualquer verificação de validade de dados inseridos através de uma declaração de entrada (INPUT).

Exercício 3

1.
```
100 INPUT X
200 INPUT Y
300 LET A = Y
```

```
400  LET Y = X
500  LET X = A
600  PRINT X, Y
700  PRINT Y, X
9999 END
```

2.
```
100  INPUT N
110  LET M = INT (N/10)
120  LET P = N — M*10
130  PRINT N, 10*P + M
9999 END
```

3.
```
100  LET A = 1017
110  LET B = 43
120  PRINT A, B, A + B + B, A + A, B — A
9999 END
```

4.
```
100  INPUT N
110  PRINT «NUMERO», «QUADRADO», «CUBO»
120  PRINT N, N*N, N*N*N
9999 END
```

5.
```
100  INPUT N
110  LET W = INT(N/4)
120  PRINT W, «CONJUNTOS DE QUATRO»
130  PRINT N — 4*W, «EM EXCESSO»
9999 END
```

6.
```
100  INPUT L, P
110  LET R = P/100
120  LET Q = (1 + R) ↑ N
130  PRINT (L*R*Q)/(12*(Q — 1))
9999 END
```

7.
```
100  INPUT X, Y, C
110  LET K = 3.141592/180
120  LET A = C*K
130  LET B = .5*X*Y*SIN(A)
140  LET E = ATN (X*SIN(A)/(Y — X*COS(A)))/K
150  LET F = ATN (Y*SIN(A)/(X — Y*COS(A)))/K
160  LET G = SOR (X*X + Y*Y — 2X*Y*COS(A))
170  PRINT «ANGULOS =», E, F
180  PRINT «AREA = », B
190  PRINT «TERCEIRO LADO = », G
9999 END
```

8.
```
100  INPUT R
110  LET A = R*57.2958
120  LET D = INT(A)
130  LET Q = 60*(A — D)
140  LET M = INT(Q)
150  LETS = INT((Q — M)*60 + .5)
160  PRINT R, «RADIANOS = », D, M, S
9999 END
```

Exercício 4

1.
```
100  LET N = 1
110  PRINT N, N*N, N*N*N
120  LET N = N + 1
130  IF N < 101 THEN 110
9999 END
```

2.
```
100  LET T = 0
110  LET K = 0
120  INPUT N
130  LET T = T + N
140  LET K = K + 1
150  IF K < 10 THEN 120
160  PRINT «MEDIA = », T/10
9999 END
```

3.
```
100 LET N = 2
110 PRINT 1/N
120 LET N = N + 1
130 IF N < 101 THEN 110
9999 END
```

4.
```
100 LET A = 5
110 REM D = MKS EM 1P, F = FRANCOS
120 INPUT D, F
130 PRINT A, A*F, A*D
140 LET A = A + 1
150 IF A < 101 THEN 130
9999 END
```

5.
```
100 PRINT «A», «B», «C»
110 LET H = 100
120 LET M = 0
130 PRINT (H + M)/100, (H + M + 10)/100,
135 PRINT (H + M + 20)/100
140 LET M = M + 20
150 IF M < 60 THEN 130
160 LET H = H + 100
170 IF H < 400 THEN 120
9999 END
```

6.
```
100 LET K = 0
110 INPUT A, B
120 LET H = SQR (A ↑ 2 + B ↑ 2)
130 PRINT A, B, H
140 LET K = K + 1
150 IF K < 0 THEN 110
9999 END
```

7.
```
100 LET P = 0
110 LET J = 0
120 LET K = 0
130 LET N = 1
140 LET T = 1
150 LET T = SGN(T)*1/N
160 LET P = P + T
170 LET N = N + 2
180 LET T = — T
190 LET K = K + 1
200 IF K < 100 THEN 150
210 LET K = 0
220 PRINT 4*P
230 LET J = J + 1
240 IF J < 10 THEN 150
9999 END
```

8.
```
100 LET L = 0
110 LET M = 0
120 LET N = L + M
130 LET L = M
140 LET M = N
150 IF N < 10 ↑ 3 THEN 120
160 IF N > 10 ↑ 6 THEN 9999
170 PRINT N
180 GO TO 120
9999 END
```

9.
```
100 REM MENSAGEM PARA OPERADOR
110 PRINT «ENTRAR SEIS NUMEROS»
120 INPUT A, B, C, P, Q, R
130 LET D = A*Q — B*P
140 LET E = B*R — C*Q
150 IF D = 0 THEN 200
160 LET X = E/D
170 LET Y = (P*C — A*R)/D
180 PRINT X, Y
190 STOP
200 IF E < > 0 THEN 230
210 PRINT «NAO INDEPENDENTE»
220 STOP
230 PRINT «INDETERMINADO»
9999 END
```

Exercício 5.1

1.
```
100  FOR N = 1 TO 100
110  PRINT N, N*N, N*N*N
120  NEXT N
9999 END
```

2.
```
100  LET T = 0
110  FOR K = 1 TO 10
120  INPUT N
130  LET T = T+ N
140  NEXT K
150  PRINT «MEDIA = », T/10
9999 END
```

3.
```
100  FOR N = 2 TO 100
110  PRINT 1/N
120  NEXT N
9999 END
```

4.
```
100  D = MARCOS EM 1P, F = FRANCOS
110  INPUT D, F
120  FOR A = 5 TO 100
130  PRINT A, A*F, A*D
140  NEXT A
9999 END
```

5.
```
100  PRINT «A», «B», «C»
110  FOR H = 100 TO 400 STEP 100
120  FOR M = 0TO60 STEP 20
130  PRINT (H + M)/100, (H + M + 10)/100,
135  PRINT (H + M + 20)/100
140  NEXT M
150  NEXT H
9999 END
```

6.
```
100  FOR K = 1 TO 10
110  INPUT A, B
120  LET H = SQR(A ↑ 2 + B ↑ 2)
130  PRINT A, B, H
140  NEXT K
9999 END
```

7.
```
100  LET N = 1
110  LET T = 1
120  LET P = 0
130  FOR K = 1 TO 10
140  FOR L = 1 TO 100
150  LET P = SGN(T)*1/N + P
160  LET T = — T
170  LET N = N + 2
180  NEXT L
190  PRINT «DEPOIS», K*100, «TERMOS: PI =»,
195  PRINT 4*P
200  NEXT K
9999 END
```

Exercício 5.2

1.
```
100  (lista de dados)
110  LET T = 0
120  DIM A(10)
130  FOR N = 1 TO 10
140  READ A(N)
150  LET T = T + A(N)
160  NEXT N
170  LET M = T/10
180  LET D = 0
190  FOR J = 1 TO 10
200  IF D > ABS (A(J) — M) THEN 230
210  LET D = ABS(A(J) — M)
220  LET G = A(J)
230  NEXT J
240  PRINT M, G
9999 END
```

2.
```
100  DIM X(5, 3)
110  LET B = X(1, 1)
```

```
120  FOR I = 1 TO 5
130  FOR J = 1 TO 3
140  IF X(I, J) < 3 THEN 160
150  LET B = X(I, J)
160  NEXT J
170  NEXT I
180  FOR I = 1 TO 5
190  FOR J = 1 TO 3
200  LET X(I, J) = X(I, J)/B
210  NEXT J
220  NEXT I
9999 END
```

3.
```
100  DIM A(3, 4)
110  LET B = A(1, 1)
120  LET I = 1
130  LET J = 1
140  FOR K = 1 TO 3
150  FOR L = 1 TO 4
160  IF A(K, L) > B THEN 210
170  NEXT L
180  NEXT K
190  PRINT B, I, J
200  STOP
210  LET I = K
220  LET J = L
230  LET B = A(K, L)
240  GO TO 170
9999 END
```

4.
```
100  DIM P(50)
110  LET I = 1
120  LET J = 3
130  LET P(1) = 3
140  LET K = 0
150  LET K = K + 1
160  IF P(K) <= INT (SQR(J)) THEN 230
170  PRINT J
180  LET P(I) = J
```



```
190  LET I = I + 1
200  LET J = J + 2
210  IF J < 100 THEN 140
220  STOP
230  LET M = INT(J/P(K))
240  IF J = P(K)*M THEN 200
250  GO TO 150
9999 END
```

5.
```
100  DIM A(10)
110  FOR M = 1 TO 5
120  LET A(M) = M
130  LET A(M + 5) = M
140  NEXT M
150  FOR N = 0 TO 4
160  PRINT   A(N + 1),   A(N + 2),   A(N + 3),
165  PRINT A(N + 4), A(N + 5)
170  NEXT N
9999 END
```

6.
```
100  (lista de dados)
110  LET L = 0
120  LET M = 1000
130  LET T = 0
140  REM L = MAIOR, M MENOR, T TOTAL
150  LET K = 1
160  REM K = CONTADOR
170  READ X
180  IF X = —1 THEN 260
190  LET T = T + X
200  LET K = K + 1
210  IF L > X THEN 230
220  LET L = X
230  IF M < X THEN 250
240  LET M = X
250  GO TO 170
260  PRINT «MEDIA», MAIOR», «MENOR»
270  PRINT T/K, L, M
9999 END
```

Exercício 5.3

1. (*a*)
```
8000 LET M = INT(X/1000)
8010 LET C = INT((X — 1000*M)/10)
8020 LET R = X — 1000*M — 10*C
8030 RETURN
```

(*b*)
```
8000 LET Z = 0
8010 REM Z = TOTAL, X ARRAY, V MEDIA
8020 FOR Q = 1 TO 10
8030 LET Z = Z + X(Q)
8040 NEXT Q
8050 LET V = Z/10
8060 RETURN
```

(*c*)
```
8000 LET Q = P
8010 REM P = PRECO
8015 REM Q = PRECO COM DESCONTO
8020 IF Q > 100 THEN 8070
8030 IF Q > 50 THEN 8050
8040 RETURN
8050 LET Q = .975*Q
8060 RETURN
8070 LET Q = .95*Q
8080 RETURN
```

(*d*)
```
8000 LET Z = E*P
8010 REM E = TAXA, P = LIBRAS, $.CC EM Z
8020 RETURN
```

2.
```
8000 LET S = A(1)
8010 REM ARRAY = A, DIMENSAO EH Z
8020 REM MAIOR EM L, MENOR EM S
8030 LET L = A(1)
8040 FOR Y = 2 TO Z
8050 IF L < A(Y) THEN 8100
8060 IF S > A(Y) THEN 8120
8070 NEXT Y
8080 PRINT «MAIOR =», L, «MENOR =», S
```



```
8090 RETURN
8100 LET L = A(Y)
8110 GO TO 8070
8120 LET S = A(Y)
8130 GO TO 8070
```

3.
```
100  INPUT N
110  GOSUB 8000
120  STOP
8000 PRINT «FACTORES DE», N, «ABAIXO»
8010 DATA 2, 3, 5, 7
8020 FOR Z = 1 TO 4
8030 READ Y(Z)
8040 NEXT Z
8050 RESTORE
8060 FOR Z = 1 TO 4
8070 LET Q = N/Y(Z)
8080 IF N < Q*Y(Z) THEN 8130
8090 PRINT Y(Z)
8100 IF Q = 1 THEN 8150
8110 LET N = Q
8120 GO TO 8070
8130 NEXT Z
8140 PRINT N
8150 RETURN
9999 END
```

4.
```
8000 LET Z = B*B — 4*A*C
8010 IF Z < > 0 THEN 8040
8020 PRINT «UNICA SOLUCAO, X = »,
8025 PRINT — B/2*A
8030 RETURN
8040 IF Z < 0 THEN 8070
8050 PRINT «X = », (—SQR(Z))/2*A, «E»,
8055 PRINT «X = », ( — B — SQR(Z))/2*A
8060 RETURN
8070 PRINT «SOLUCOES IMAGINARIAS»
8080 RETURN
```

5.
```
8000 LET Z = A*57.2958
8010 REM RADIANOS EM A
8020 REM GRAUS EM D, MINUTOS M
8025 REM SEGUNDOS EM S
8030 IF Z <= 360 THEN 8060
8040 LET Z = Z — 360
8050 GO TO 8030
8060 LET D = INT(Z)
8070 LET Z = (Z — D)*60
8080 LET M = INT(Z)
8090 LET S = INT((Z — M)*60 + .5)
8100 RETURN
```

6. Os exercícios a — f têm o seguinte programa principal comum:

```
100  (lista de dados com 3 elementos)
110  DIM E(3)
120  FOR I = 1 TO 3
130  READ E(I)
140  PRINT FNA E(I)
150  NEXT I
9999 END
```

(*a*) 400 DEF FNA(X) = LOG(X)/LOG(10)
(*b*) 400 DEF FNA(X) = TAN(X*.0174533)
(*c*) 400 DEF FNA(X) = 3.141593*X ↑ 2
(*d*) 400 DEF FNA(X) = X*.0174533
(*e*) 400 DEF FNA(X) = INT(X + .5)
(*f*) 400 DEF FNA(X) = X ↑ .3333333

Exercício 6

1.
```
100  lista de dados
110  DIM A$(5), B$(5)
120  LET D$ = «ZZZZZZ»
130  FOR N = 1 TO 5
140  READ A$(N)
150  NEXT N
160  FOR M = 1 TO 5
170  LET E$ = D$
180  LET L = 1
190  FOR N = 1 TO 5
200  IF A$(N) >= E$ THEN 220
210  LET E$ = A$(N)
220  LET L = N
230  NEXT N
240  LET A$(L) = D$
250  LET B$(M) = E$
260  NEXT M
270  FOR K = 1 TO 5
280  PRINT B$(K)
290  NEXT K
9999 END
```

2.
```
100  INPUT N
110  DATA «ZERO», «UM», «DOIS», «TRÊS»
115  DATA «QUATRO»
120  DATA «CINCO», «SEIS», «SETE», «OITO»
125  DATA «NOVE»
130  DATA «DEZ»
140  DIM X$(11)
150  FOR K = 1 TO 11
160  READ X$(K)
170  NEXT K
180  LET R = N — 11*INT(N/11)
190  PRINT X$(R + 1)
9999 END
```

3.
```
100  (lista de dados)
110  FOR L = 1 TO 10
120  READ N
130  PRINT N,
140  FOR M = 1 TO 10
150  PRINT «+»;
160  NEXT M
170  PRINT
```

```
180 NEXT L
9999 END
```

4.
```
100 REM LIMPAR CONTADORES 1—6
110 DIM N(6)
120 FOR K = 1 TO 6
130 LET N(K) = 0
140 NEXT K
150 FOR M = 1 TO 100
160 LET X = INT(6*RND(1)) + 1
170 LET N(X) = N(X) + 1
180 NEXT M
190 REM IMPRIMIR CONTADORES 1 — 6
200 FOR K = 1 TO 6
210 PRINT N(K)
220 NEXT K
9999 END
```

5.
```
100 REM NUM. ALEATORIO INDICA VALOR
105 REM DOS NAIPES
110 REM COPAS 1-13, OUROS 14-26; PAUS 27-39;
115 REM ESPADAS 40-52
120 REM TAL QUE 1 PAUS = 33
130 FOR K = 1 TO 13
140 LET X = INT(52*RND(1)) + 1
150 REM CALCULAR INDICADOR DE NAIPE
160 LET A = INT(X/13.3)
170 LET B = X — 13*A
180 REM B = POSICAO EM NAIPE
190 IF B = 1 THEN 300
200 IF B = 12 THEN 320
210 IF B = 13 THEN 340
220 IF B = 11 THEN 360
230 PRINT B, «DE»
240 IF A = 0 THEN 380
250 IF A = 1 THEN 400
260 IF A = 2 THEN 420
270 PRINT «ESPADAS»
280 NEXT K
290 STOP
300 PRINT «AS DE»
310 GO TO 240
320 PRINT «DAMA DE»
330 GO TO 240
340 PRINT «REI DE»
350 GO TO 240
360 PRINT «VALETE DE»
370 GO TO 240
380 PRINT «COPAS»
390 GO TO 280
400 PRINT «OUROS»
410 GO TO 280
420 PRINT «PAUS»
430 GO TO 280
9999 END
```

6.
```
100 PRINT «PAR», «IMPAR», «MANQUE»,
105 PRINT «PASSE»
110 REM LIMPAR CONTADORES
120 LET A = 0
130 LET B = 0
140 LET C = 0
150 LET D = 0
160 FOR K = 1 TO 1000
170 LET X = INT(36*RND(1)) + 1
180 IF X < 19 THEN 250
190 LET D = D + 1
200 IF X — 2*INT(X/2) = 0 THEN 270
210 LET B = B + 1
220 NEXT K
230 PRINT A, B, C, D
240 STOP
250 LET C = C + 1
260 GO TO 200
270 LET A = A + 1
280 GO TO 220
9999 END
```

GLOSSÁRIO

Esta é uma lista de termos que são utilizados neste livro e em outras obras sobre a programação de sistemas microcomputadores. Ela não inclui termos que digam apenas respeito à construção electrónica e à arquitectura de microcomputadores e tenham pouca relevância para a programação. Termos que se apliquem apenas aos grandes computadores não foram incluídos.

Acumulador. Uma posição de memória dentro do computador onde devem ser armazenados os dados para que se possam efectuar operações aritméticas e lógicas sobre eles.

Apontador de pilha *(stack pointer)*. Um registo que define a posição actual da pilha, ou seja, o endereço do último elemento a dar entrada na pilha.

Aritmética binária. Aritmética executada sobre operandos binários.

Array. Ver *Quadro*.

ASCII. Abreviatura de *American Standard Code for Information Interchange*, que é o código mais vulgar de representação de caracteres alfanuméricos em dígitos binários dentro de um sistema microcomputador.

Assembler. Programa que converte um programa escrito em linguagem «assembler» *(assembly language)* em código-máquina. A linguagem «assembler» é uma linguagem de programação que utiliza mnemónicas para facilitar a programação ao nível do código-máquina. Geralmente não se faz a distinção entre o nome da linguagem e o programa que a vonverte.

BCD. Abreviatura de *binary coded decimal* (decimal codificado em binário). É uma forma de exprimir cada dígito decimal através de quatro dígitos binários dentro dum sistema microcomputador.

Binário. Sistema de representação de números através dos dois dígitos binários 1 e 0.

Bit. Abreviação de *binary digit* (dígito binário).

Bloco *(block)*. Número de items de informação transferidos por uma única instrução de um dispositivo de ficheiro tal como uma diskette ou uma fita de cassette. Utiliza-se igualmente este termo para designar a área física destes dispositivos onde se encontram gravados os dados.

Bug («piolho») Palavra utilizada na literatura técnica anglo--saxónica para designar um erro num programa (ver também *Debug)*.

Bus. Interconexão dentro de um sistema microcumputador na qual os dados passam de uma parte do sistema para a outra. Os «buses» mais importantes são: *data bus*, para transferências de dados dentro do sistema e de traço para periféricos; *control bus,* que transporta sinais para a operação do sistema; *adress bus,* que transporta os 16 bits do endereço de identificação dos dados.

Byte. Um grupo de bits tratado como uma unidade. Contém geralmente oito bits.

Campo *(field)*. O número de bits ou bytes necessários para conter um item de dados específico.

Chip. Ver *Pastilha.*

Clock *(relógio)*. Oscilador que gera os impulsos de tempo para o CPU.

Código-máquina *(machine code)*. A linguagem de programação básica que um sistema microcomputador pode obedecer sem a utilização de um interpretador ou de um compilador.

Código de operação. São os bits de uma instrução que determinam o processo a executar pela instrução.

Compilador. O programa de computador que traduz a totalidade de um programa escrito numa linguagem de alto nível para código-máquina.

Contador de programa *(program counter)*. O registo que contém o endereço da próxima instrução a ser executada.

Core. Memória do computador.

CPU. Abreviatura de *Central Processing Unit.* Unidade central de processamento (UPC) de um computador que contém as unidades aritmética e lógica onde são executadas as instruções dum programa.

Cursor. Traço de luz num écran VDU que indica onde vai aparecer o próximo carácter (por digitação ou por programa).

Debug. Processo de eliminação dos erros de programação (ver também *Bug)*.

Disco *(disk)*. Disco magnético sobre o qual são gravados ficheiros de dados. Um sistema microcomputador pode ler ou gravar dados em discos.

Diskette. Pequeno disco magnético. Também designado por *floppy disc* (ver em baixo).

Endereçamento indexado *(indexed adressing)*. A forma de endereçamento em que o endereço directo é encontrado pela adição do conteúdo de um registo (conhecido por *registo de índice)* ao valor armazenado no próprio comando.

Endereço *(adress)*. A referência que identifica uma posição de memória

Endereço directo *(direct adress)*. Por vezes chamado *endereço absoluto*. É a identificação permanente, standard, de uma posição de memória.

Endereço imediato *(immediate adress)*. A forma de endereçamento em que um operando é armazenado no próprio comando após o código de instrução.

EPROM. Abreviatura de *Electrically Programmable Read-Only Memory.* Uma parte da memória do sistema que não pode ser destruída por sobreposição de escrita, mas que pode ser apagada (geralmente através de luz ultravioleta) e escrita de novo.

Firmware. Instruções ou dados gravados permanentemente.

Flag. Um bit que pelo seu valor (1 ou 0) significa se uma dada condição se verifica ou não.

Floppy disc. Disco magnético flexível (ver também *Diskette*).

Fluxograma (*flowchart*). Representação gráfica de um programa de computador.

Hardware. Componentes electrónicos e mecânicos de um sistema computador.

Hexadecimal. Sistema de numeração de base 16, que utiliza os dígitos 0 — 9 e as letras A — F para representação dos números.

Interface. Um dispositivo, e os programas apropriados, que permitem um determinado dispositivo electrónico (tal como um periférico) comunicar com o CPU.

Interpretator (*interpreter*). Um programa que traduz declarações de uma linguagem de alto nível em código-máquina, sendo cada declaração executada pela máquina à medida que é traduzida.

Interrupção. Um sinal que provoca a suspensão da execução de um programa, de forma a ser executada uma secção de programa que trata do dispositivo ou da condição que deu origem ao sinal de interrupção.

Instrução lógica. Uma instrução que executa operações lógicas «e» e «ou» num sistema microcomputador. Utilizada por vezes para realizar deslocamentos (*shifts*) a fim de ignorar o sinal de um byte ou palavra.

K. Uma medida da dimensão da memória de um computador: 1024 bytes ou palavras.

Label. Ver *Rótulo*.

LED. Abreviatura de *Light-emitting diode* (díodo emissor de luz). Um tipo de mostrador que se encontra com frequência nos relógios digitais e calculadoras electrónicas.

Linguagem de alto nível (*hig-level language*). Uma linguagem de programação orientada para o problema e não para a máquina. Utiliza a língua inglesa e a notação matemática, que é depois traduzida para código-máquina por um interpretador ou um compilador. Linguagens de alto nível: Algol, Basic, Fortran, Cobol, Pascal, APL, PL/I, etc.

Máscara (*mask*). Um padrão de bits utilizado em instruções lógicas para seleccionar um padrão de bits determinado de um byte ou palavra.

Memória. A parte do sistema que armazena os dados e as instruções de programa, de onde podem ser recuperadas em alta velocidade, independentemente da sua posição.

MPU. Abreviatura de *Microprocessor Unit*. É o CPU de um microcomputador.

Octal. Sistema de numeração de base 8, que utiliza os dígitos 0 — 7 para representação de números.

Operando. A parte de endereço de uma instrução. Este termo é igualmente utilizado para os dados que intervêm numa operação de computador.

Palavra (*word*). Um grupo de bits tratado como uma unidade a que é dada uma posição única de memória.

Pastilha (*chip*). Placa de sílica sobre a qual é construído um circuito integrado.

Periférico. Um dispositivo externo de entrada-saída (*input-output*) de dados.

Programa. Um conjunto de instruções para um sistema microcomputador executar uma tarefa definida.

Programa fonte (*source program*). Um programa que tem de ser traduzido para código-máquina.

Programa objecto (*object program*). O programa em código-máquina produzido por um assembler ou um compilador.

Pilha (*stack*). Uma parte da memória em que os dados são armazenados segundo o princípio LIFO (*«last in, first out»*): O último a entrar é o primeiro a sair.

PROM. Abreviatura de *Programmable Read-only Memory*. Uma forma especial de ROM que pode ser programada pelo utilizador (vem também *ROM*).

Quadro (*array*). Um conjunto de valores que são colectivamente conhecidos por um nome único Os valores individuais são referenciados pelo nome do quadro e por um índice ou subscrito que indica a sua posição dentro do quadro: por

exemplo, B(2) é o segundo elemento do quadro unidimensional (ou *vector*) B; X(1,4) é o quarto elemento da primeira linha do quadro bidimensional (ou *matriz*) X. Listas e tabelas constituem formas vulgares de quadros.

RAM. Abreviatura de *Random-access Memory* (memória do acesso aleatório). O tipo vulgar de memória de computador; um programador pode ler ou escrever nesta memória, e qualquer que seja a posição de memória em que os dados se encontrem a velocidade da sua recuperação é a mesma.

Registo *(register)*. Uma posição de memória privilegiada onde o acesso pelo CPU é extremamente rápido.

ROM. Abreviatura de *Read-only Memory* (memória exclusiva de leitura). Memória que não pode ser alterada pelo programador.

Rótulo *(label)*. Uma identificação de uma posição de memória.

Software. Programas de computador. Utilizado frequentemente para designar o conjunto de programas fornecido pelos construtores de sistemas computadores.

Stack. Ver *Pilha*.

Subrotina. Uma sequência de instruções de programa que é escrita apenas uma vez e que pode ser chamada muitas vezes dentro de um programa.

VDU. Abreviatura de *Visual Display Unit*. Ecran tipo-televisão para a apresentação de dados. Alguns microcomputadores podem ser ligados a vulgares televisores para utilizarem o seu tubo de raios catódicos.

Vírgula flutuante *(floating point)*. Um método de representação de números que possuem parte fraccionária. Representação E em muitas calculadoras electrónicas.

ÍNDICE

PREFÁCIO ... 7

1 — INTRODUÇÃO À PROGRAMAÇÃO DE COMPUTADORES ... 9

- Verifique se o problema pode ser resolvido pelo computador ... 12
- Definição do problema a programar ... 14
- Como fazer fluxogramas ... 15
- Documentação ... 31

2 — ESCOLHA DE UMA LINGUAGEM DE PROGRAMAÇÃO ... 33

- Código-máquina ... 34
- Linguagem assembler ... 37
- Linguagem de alto nível ... 40
  - *Basic* ... 43
  - *Cobol* ... 44
  - *Fortran* ... 44
  - *Pascal* ... 45
  - *Algol* ... 45
  - *PL/I* ... 46
  - *APL* ... 46

3 — INTRODUÇÃO AO BASIC ... 47

- Entrada de programas ... 47
- Comandos do sistema ... 50
- Números de linha ... 51
- Comentários ... 53
- A declaração LET ... 54
- Nomes das variáveis ... 55
- Números ... 56
- Operadores aritméticos ... 58
- Combinação de operações aritméticas (expressões aritméticas) ... 59

Ordem de cálculo das fórmulas (expressões aritméticas) ... 60
Funções ... 63
Entrada simples de dados ... 66
Entrada de números durante a execução do programa 68
Impressão de texto ... 69
Impressão simples ... 69
Um programa completo ... 70
Exercícios 3 ... 71

4 — OPERAÇÕES REPETITIVAS EM BASIC ... 73

A declaração IF-THEN ... 74
Tenha cuidado ao usar o símbolo de comparação «=» 75
A declaração STOP ... 77
Usos mais importantes da declaração IF ... 78
A declaração GO TO ... 80
Evite GO TO sempre que possível ... 81
Exercícios 4 ... 83

5 — CICLOS, LISTAS E SUBROTINAS EM BASIC ... 85

A declaração FOR ... 85
Alguns problemas que podem surgir nos ciclos FOR 89
Execução de ciclos no interior de outros («nesting») 90
Velocidade de execução dos ciclos ... 92
Exercício 5.1 ... 93
Listas e quadros ... 93
Exercício 5.2 ... 96
Funções e subrotinas ... 97
Subrotinas ... 99
Exercício 5.3 ... 102

6 — ALGUMAS POSSIBILIDADES ADICIONAIS DO BASIC ... 105

Tratamento de caracteres ... 105
Outras possibilidades de saída ... 109
Ficheiros ... 112
A função RND ... 115
Outras possibilidades ... 117
Exercício 6 ... 118

7 — VARIANTES DE BASIC ... 119

8 — INTRODUÇÃO À PROGRAMAÇÃO EM LINGUAGEM ASSEMBLER E EM CÓDIGO-MÁQUINA ... 129

Alguns aspectos fundamentais ... 131
Transferência de dados ... 133

Transferências entre a memória e o acumulador 134
Transferências de dados envolvendo outros registos ... 136

Adição e subtracção ... 139
Aritmética de multi-precisão ... 144
Adição e subtracção de decimais em codificação binária ... 147

9 — MAIS POSSIBILIDADES DA LINGUAGEM ASSEMBLER E DO CÓDIGO-MÁQUINA ... 151

«Branching» ... 151
Ciclos ... 155
Arrays e tabelas ... 158
Subrotinas ... 159
Operações de empilhamento ... 161
Manipulação de bits ... 164
Operações lógicas ... 167
Representação de caracteres ... 169
Entrada e saída ... 172
Interrupções ... 174

10 — DESENVOLVIMENTO E TESTE DE PROGRAMAS 179

Fases do teste de um programa codificado ... 181

Verificação das instruções ... 183
Eliminação dos erros «gramaticais» ... 183
Testes da execução do programa ... 185

SOLUÇÕES SUGERIDAS PARA OS EXERCÍCIOS ... 191

Exercício 3 ... 191
Exercício 4 ... 193
Exercício 5.1 ... 196
Exercício 5.2 ... 197
Exercício 5.3 ... 200
Exercício 6 ... 202